GraphIPC

Non-Linear Process Communication

Applied to Computational Linguistics Workflows

John Edward Vidler *MSci.*

School of Computing and Communications

Lancaster University

Abstract

Sequential operations are core to the operation of the modern computer system; a situation not unfamiliar to any computational linguist is where sequential sub-operations are performed to transform data input into meaningful information.

While the hardware available to the researcher has increased in capability and complexity greatly since its inception, the tools and systems available to manage and control this hardware has changed relatively little.

Years of legacy applications and systems may have further pushed newer systems to continue the linear nature of the communication structures available for either familiarity, or backwards compatibility.

With the advent of GPU computing and the continual increase in processor core count (both real and virtual) there is a pressing need to reexamine how these resources can be better utilised, before subsequent innovations in both hardware and software end up 'pigeon-holed' by the current mechanisms.

High-level applications have taken the normal interaction methods and redefined their action through the use of middleware frameworks and similar software solutions to present a different abstraction. This can most clearly be seen in the prevalence of star, grid and mesh networks for high performance and distributed computing.

Taking what has been learned from these disciplines, here I present a design and proof-of-concept implementation for addressing and communicating with multiple processes, allowing graph form inter-process messaging in Linux.

As the industry is currently focused on natural language processing for interaction and data analysis, I run test workloads from this set to evaluate the utility and options therein using this approach.

Declaration

I declare that this thesis is my own work and that it has not been submitted in substantially the same form for the award of a higher degree elsewhere.

The following publications have been used wholly or in part as part of this thesis, and are the author's original work:

- John Vidler et al. "Dealing With Big Data Outside Of The Cloud: GPU Accelerated Sort". In: *Challenges in the Management of Large Corpora. CMLC-2*. Reykjavik, 2014, p. 21. URL: http://www.lrec-conf.org/proceedings/lrec2014/workshops/LREC2014Workshop-CMLC2Proceedings-rev2.pdf
- John Vidler and Stephen Wattam. "Keeping Properties with the Data CL-MetaHeaders-An Open Specification". In: *CMLC-5+BigNLP* (2017). URL: https://ids-pub.bsz-bw.de/frontdoor/index/index/docId/6243

John Edward Vidler
MSci. Computer Science
1st July 2024

To my wife, Elizabeth, who for some reason stuck with me for 7 years of PhD research.

Also to my unborn daughter, whom I look forward to getting to know.

Acknowledgements

I would like to express my gratitude for all the help, suggestions and support from Paul Rayson and Andrew Scott; my supervisors for this work. Without their help and guidance throughout, I would never have finished it. I would also like to mention Utz Roedig, for his support in honing the latter stages of the project.

Thank you to my friends; Stephen Wattam, John Hardy, Carl Ellis, Christopher Bull and Martin Bor for their encouragement, support, jibes and competition both implied and explicit throughout.

Thank you to everyone at the School of Computing and Communications, throughout the years for being there to talk over coffee, bemoan the world and to offer a sympathetic ear; there are far too many of you to name, but know that you all are appreciated.

Finally thank you to my parents, for giving me the confidence to start this journey, and the support to keep me going.

Thank You All.

John Vidler

Lancaster, February 2020

Terminology

Connectome	A connectome or node-edge graph, consists of a series of nodes, normally represented as points, connected together with edges represented as lines. See also graph
Continuous Flow	Data flows in which each message is dependant in some way on the timing, or order of the messages. These flows must be processed in-order and separately unless the data itself encodes the order therein.
Data Flow	A sequence of packets along a given connection
Discrete Flow	Data flows in which each message, or packet is processed individually, without information from any other. These flows can be processed with interleaved data
Edge	A single connection between nodes in a node-edge graph, see also connectome
Graph	A connectome or node-edge graph, consists of a series of nodes, normally represented as points, connected together with edges represented as lines
Node	A single point in a node-edge graph, see also connectome
Client	A device or process where the primary mode of interconnection is outbound - connecting to servers or other clients
Mangle	A data and packet term relating to the meta-data having been mutated. A mangled packet might have its destination address altered, for example. Also see unmangle
P2P	See *Peer-to-Peer*
Peer-to-Peer	A method of interconnection whereby each device or process behaves both as a server and a client, allowing both inbound and outbound connections to be made.

Router	A device or process responsible for storing and forwarding messages sent to it along assorted connections associated with it. Frequently also performs some filtering or path-decision logic as part of this operation
Server	A device or process where the primary mode of interconnection is inbound - connections are made from other devices or processes.
Unmangle	A data and packet term relating to cases where the meta-data associated with a given payload is preserved unchanged. Also see mangle
Compute Device	Generally a processor of some kind, executing code, but also includes any device performing one or more functions on data, such as a dedicated packet processing chip.
Load Surface	A dynamic, multi-dimensional space in which each axis represents one constraint under which a compute device, or collection of compute devices operates. The surface is likely manifold, but may not be if there are unknown factors.
SoC	System-on-Chip; multiple units, which would traditionally have been distinct components on a PCB, integrated onto a single die via VLSI techniques
VLSI	Very-Large-Scale Integration; the process by which millions of distinct transistors are combined to form a single chip via (usually) photolithography.

Publications

Dealing With Big Data Outside Of The Cloud: GPU Accelerated Sort

Influences This paper demonstrated the utility of using many small processors to execute complex workloads, even if the individual processes were generally considered sub-optimal; effectively a 'many processors make light work' solution space. This design, and the discussions it generated lead directly to looking at ways of building systems to support this kind of processing, and ultimately GraphIPC itself.

Full Citation John Vidler et al. "Dealing With Big Data Outside Of The Cloud: GPU Accelerated Sort". In: *Challenges in the Management of Large Corpora. CMLC-2*. Reykjavik, 2014, p. 21. URL: http://www.lrec-conf.org/proceedings/lrec2014/workshops/LREC2014Workshop-CMLC2Proceedings-rev2.pdf

Keeping Properties with the Data CL-MetaHeaders - An Open Specification

Influences In the paper we describe a scheme for encoding the properties of corpus data for either storage or transfer; this used characteristics of the existing formats to provide a robust way for reading processes to determine the type of data held therein. Discussions spawned from this work influenced my thoughts around the general interoperability of the tools being used, and heavily influenced the overall philosophy of GraphIPC.

Full Citation John Vidler and Stephen Wattam. "Keeping Properties with the Data CL-MetaHeaders-An Open Specification". In: *CMLC-5+BigNLP* (2017). URL: https://ids-pub.bsz-bw.de/frontdoor/index/index/docId/6243

LoRa for the Internet of Things

Influences In this paper, the exploration of light-weight protocols influenced the general protocol design for GraphIPC.

Full Citation Martin Bor, John Edward Vidler and Utz Roedig. “LoRa for the Internet of Things”. In: 16 (2016), pp. 361–366. URL: https://www.researchgate.net/profile/John_Vidler2/publication/297731094_LoRa_for_the_Internet_of_Things/links/56e1893e08ae4bb9771ba9e3/LoRa-for-the-Internet-of-Things.pdf

It Bends but Would it Break? Topological Analysis of BGP Infrastructures in Europe

Influences In this work, we examined the large-scale internet infrastructure in the UK internet. The analysis of this network lead to more investigation in general around how interconnected systems behaved, and where the critical points were.

Full Citation Sylvain Frey et al. “It Bends but Would it Break? Topological Analysis of BGP Infrastructures in Europe”. In: *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2016, pp. 423–438. URL: https://eprints.soton.ac.uk/412811/1/EuroSnP.pdf

Shapeclip: Towards rapid prototyping with shape-changing displays for designers

Influences My contributions to this project were centered around the communications protocols and methods used to control the devices. As this used a display and light-dependent resistors to realise this channel, the bandwidth available was extremely small, leading me to investigate lightweight protocols and communication designs, which would later effect my work on the protocol for GraphIPC.

Full Citation John Hardy et al. “Shapeclip: towards rapid prototyping with shape-changing displays for designers”. In: *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. ACM. 2015, pp. 19–28

Contents

Chapter 1

Introduction

With the ever increasing complexities of analysis and processing techniques available for research (and other) purposes, the limitations of existing models of communication between applications are becoming clear. Rather than address these through radical restructuring of the entire operating system, as has been posited previously, the design and prototype outlined and developed here describes a complementary system, which may be included alongside completely standard systems as well as potentially being embedded as part of the system itself (with some advantages therein).

As will be discussed in the subsequent chapters of this thesis; fundamentally, the defining component of the modern application or process is that of *communication*. Without good internal and external communication infrastructure, there will always be a limit to the complexity - and thus utility - of the applications that can be described.

Communications is key in the efficient machinations of the modern system.

First, however, before the details of the design of communication systems can be covered, some description of the changes in the very nature of computing is necessary to frame the rest of the discussion.

1.1 The Change

Perhaps the easiest aspect of these changes to be understood is that of the hardware, as there has been a clear increase in local compute power and memory density that has been progressing since the inception of the technology itself. This overall increase in compute capability has generally followed Moore's Law, although in the last few years there has been discussion involving a "Post-Moore's Law" state[1]. This increase in computing capability now presents a new class of problem - that of being able to transfer data in and out of the processing elements fast enough to realise their actual performance.

This has most easily been seen in the field of General-Purpose Graphics Processing Unit

(GPGPU) research, where until recently, most of the techniques employed were limited by the rate at which data in main memory (RAM) could be transferred into the graphics card memory (or memories, in the case of linked GPU configurations). This, in conjunction with a 'batch-mode' general style of operation quickly lead to execution times entirely dominated by the import and export phases.

Thanks to innovation in computer game graphics technology, this workflow is now waning - the introduction of multiple 'Copy Engines'[2] has greatly sped up the available throughput, but also the addition of streaming techniques; whereby the data is just-in-time loaded while other processing is underway - and dynamic execution of threads on the cards themselves can, if employed carefully, entirely eliminate the initial data import delays.

1.2 Networks, External and Internal

With networking as the topic at hand, it is worth noting the developments in the internal and external wiring present in the modern system.

At the outset, networking between hosts was affected by circuit-switched technologies, effectively directly linking host to host, which quite quickly presented numerous problems at scale. These problems were largely solved by moving to a packet-switched network design, with individual point-to-point links only going so far as the first 'hop' in long network paths.

The real benefit of adopting the scheme, however, was that it became a largely stateless operation to send data between hosts; intermediate devices could merely read the data in received packets and immediately determine the operations required - normally forwarding along the next point-to-point link in the network towards the destination host.

The addition of *routing logic* to this design enabled the networks we enjoy today, with features such as Network Address Translation (NAT), firewall rules and others defining and shaping the boundaries of the networks each host can observe. The further additions of protocols such as Multi-Protocol Label Switching (MPLS) and later additions to the underlying routing such as *multicast* further enabled the logical grouping and marshalling of data in the networks.

The hardware upshot of this is that network routers (and other devices) are now extremely capable computing platforms in their own right; capable of handling messages at gigabit or greater speeds with ease[22, 23, 43, 56]. If one looks at such devices as a collection of their capabilities, rather than just as a router, it is this author's opinion that they must be regarded as not dissimilar to a co-processor themselves, specialising in high-throughput stateless message handling.

1.3 Users and Usage

In hand with the changes in the platforms and networks (concepts that are increasingly difficult to separate), the uses that computers are employed for has changed in kind.

The early mainframe-based or mainframe-styled computers attempted to meter out limited resources to numerous jobs from a (comparatively) large number of concurrent users[3] - this is in stark contrast to the systems available today, where individual users have a plethora of processors and vast amounts of working memory to use.

The very nature of the jobs (tasks, or processes) being asked of the computer has also changed. Frequently, processes are not a single run-time context to be executed to a completion state and dismissed, but are complex, interconnected networks of processes, sometimes (or indeed, oftentimes) spanning multiple hosts, and indeed, multiple networks[14, 40, 51]. This *network oriented* system view is not limited to applications requiring the use of multiple hosts (thus spanning the 'external' network) but actually includes the *internal* network of the modern host, with elaborate co-processor devices[50, 52] acting as smaller, yet complete hosts unto themselves executing within the larger (and more traditionally defined) host.

Run-time requirements and dependencies of this new form of process vary with time and environment, so are difficult to predict, sometimes leading to patterns of *migration* around the system (or collection of systems), further increasing the complexity of the *load surface* the compute resources are subject to.

1.4 Operating Systems and Resources

While clearly advantageous to the user, this glut of resources brings its own challenges in managing *how* to use them.

Examining memory as an example, the additional working memory available to the system means that larger working sets can be ready to go for when the processes running require it, but conversely, the larger working sets mean that the system itself has to expend more resources tracking, loading, and unloading them (both in terms of execution time, and that of operational complexity). This, unfortunately, may lead to 'stuttering' (also known as 'jitter') whereby the system itself has to pause executing user processes to perform the maintenance tasks required to *continue* running the user processes. Obviously, this 'stuttering' is a relatively extreme case, and thankfully rarely seen in normal execution, but lower resourced devices running heavyweight code often see this effect, nonetheless.

The heterogeneity of the modern compute platform also proves problematic. An operating system running on a platform with dissimilar processing elements has the the task of determining exactly *where* to execute the new process - a complex task considering that the trade-offs

between the processors may mean that they result in similar, or identical probability of their use.

Some operating systems have attempted to handle this by encoding the requirements of a process into the code for the process itself (or the loader, preamble, or other associated meta-data along with the binaries for the process), but even then, running the same process multiple times immediately creates compute-resource contention problems that need further solving.

Specific examples of this being done in research applications will be discussed in Chapter 2, in which a notable kernel design approach is that of the 'Capability System'[47].

As has already been mentioned; there have been a number of attempts to address some or all of these problems through the complete re-engineering of the operating system itself. This is in contrast to approaches employing libraries to mediate between the system itself and the processes running on them - these *middleware* systems use knowledge specific to their particular task niche to determine (what the library thinks is) the best execution environment for the jobs en-queued.

Whereas the systems-based approaches have focused on the nature of the interactions at the system level, some of the middleware frameworks have evolved to focus on the *workflow* of the users operating the system. These frameworks - some of which will be explored in detail in the next chapter - highlight the system interactions with the user and their processes, and in particular, the requirements for communications between said processes.

But it should be noted here that these two aspects - the workflow and the operating system itself - can no longer be considered in isolation, as network semantics reach down into the system, and up from the hardware into the execution environment, it is paramount that the capabilities at all levels reflect the needs of the processes being executed on them.

1.5 Research Questions

RQ1 What would a modern workload-focused approach look like for local message passing?

RQ2 What affordances are there to use the modern heterogeneous systems more effectively for analysis tasks?

RQ3 What workflow elements are required to support large scale data and text analytical tasks?

RQ4 What of systems design attempts from the last 30 years is actually applicable or appropriate for modern workloads?

1.6 Structure of this Thesis

The remainder of this document are divided into the following topics; first, the background material in both the operating systems area and in an application space; corpus linguistics

(Chapter 2), then progress to define the characteristics of the space between the processes and their interactions (Chapter 3). Beyond this, the remaining pages detail an idealised design (Chapter 4), along with the practical implementation limitations imposed on this design by existing hardware (Chapter 5). The prototype is then evaluated for correctness against a number of example workflows and data collections (Chapter 6), with the results discussed in the chapter thereafter (Chapter 7) when put against the original model.

Footnotes

[1]Moore's Law, which states that *transistor density* doubles every 18 months - affecting both processing capability and memory density - has seen the actual density beginning to deviate from the predicted density, showing a generally decelerating trend.

[2]A general term for subsystems of the hardware designed to ship data from place to place, often daughter-board to mother-board or back.

[3]Concurrent requests, although frequently these were handled as sequential jobs, rather than true parallel operations due to the limitations of the hardware and operating systems of the time.

Chapter 2

Related Work

To understand the need for such a system as proposed and demonstrated in this document, one must first establish two things; firstly, the nature of the processing workflow employed at the time of writing and secondly, the nature of the systems available to execute these processes upon.

As analysis and processing techniques have developed and improved, they have naturally had an a corresponding increase in complexity. This complexity is manifest both in the capabilities of the processes in question, and also in the interactions *between* the processes used.

As tools grow, there reaches a point where the tool is actually performing many separate jobs and processing them in combinations; in an effort to reduce internal complexity of such tools, it is often the case that they are refactored into a number of smaller, single-purpose sub-tools, each which communicates with the next via a communication link; the specific nature of these links will shortly be discussed at length, but may be signals, files, sockets, or any other mechanism provided for this purpose.

2.1 The GATE Toolkit

If the reader is familiar with tooling common to the corpus linguistics (CL) and natural language processing (NLP) fields, it is probably no surprise that at this stage there is a clear parallel with the work of Cunningham and their GATE toolkit[12].

GATE, or the *General Architecture for Text Engineering* provides a combined wrapper and dependency modelling system for NLP tools - this allows users to describe the operations they need in a graphical, node-graph format, then have the system handle the order of operations.

Processes are executed serially in order to satisfy the requirements of subsequent operations, at which stage the next set of operations can be executed (See 2.1 for a block diagram of GATE's internals). GATE does this by having a declaration language for input requirements and output productions, from which the toolkit can infer the operational order and requirements

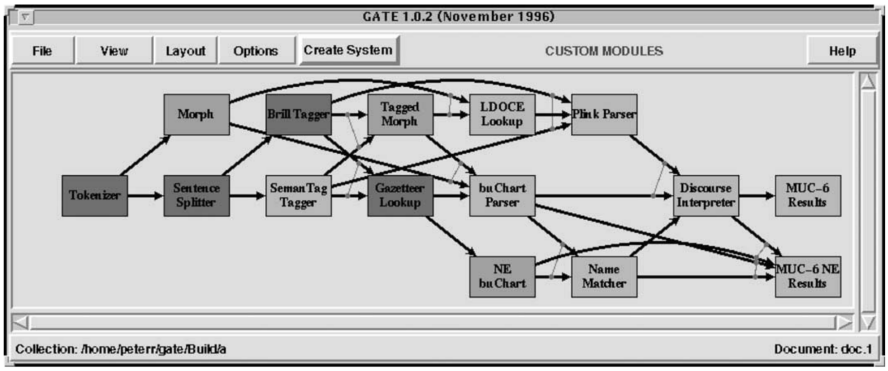


Figure 2.1: The internal architecture of GATE. Of note is the translation layers between tools. These translations can be done automatically provided that GATE knows the nature of the tool in question.

of each step of the graph.

This falls into the *pseudo-parallelism* model, a familiar concept in unicore systems development, whereby the system gives the illusion of running concurrent processes when in reality it simply interleaves them running one at a time - a design that will be discussed in further detail presently in this chapter.

The only nuance from the 'systems' meaning of the term here is that *each process is run to completion* rather than truly interleaved with sibling processes as would be the case in a traditional preemptive multiprocessing system, for example[4].

2.2 Use Cases

As part of their work on GATE, Cunningham *et.al*, identified 20 use cases for Language Engineering (LE)[5] that cover every aspect of the field from the individual tool operations to the workflows employed by users.

Many of these use cases are specific to particular language engineering operations, and as such do not directly relate to the discussion here, but beyond the simple cases of singly executing processes, all the use cases presented become dependant on a smaller number of process management ones.

More specifically, the *distributed processing*, *parallel processing*, and *deployment* use cases combine and impact with the remainder as the connected network of processes becomes larger. In addition to these basic cases, a further two new use cases, *reproducibility*, and *reuse*, along with a third; *workflow* should be represented here as a more general representation of user intent, as we describe a way to define a sequence of operations as a graph-flow, rather than as a simple sequence.

2.2.1 Modular NLP Tools

In addition to the work done with GATE, there are a number of tools which attempt to present each stage of a workflow as an entirely independent tool or binary. OpenNLP[6] and NLTK [6][7] are two particularly popular ones, with both offering a command-line method to access their functionality, as well as the option to use an API to directly embed the tools into other works.

Individually, each tool presents a single operation (Tagging, sentence identification, lemmatisation, and so forth) and generally reads an input source via the standard input stream, reporting the results directly to the standard output stream (an example of which can be seen in Figure 2.2). This allows the tools to be chained together into a series of operations describing a more complete workflow, as presented in Figure 2.3.

```
$> opennlp POSTagger opennlp-models/en-pos-maxent.bin < testcorpus.txt
Loading POS Tagger model ... done (0.588s)

This_DT eBook_NN is_VBZ for_IN the_DT use_NN of_IN anyone_NN anywhere_RB at_IN no_DT cost_NN and_CC with_IN
almost_RB no_DT restrictions_NNS whatsoever._WP You_PRP may_MD copy_VB it,_RB give_VBP it_PRP away_RB or_CC
```

Figure 2.2: An example of the use and initial output of OpenNLP running the POSTagger module with a pre-trained model.

Strangely, the individual command line tools are presented as proof of concepts rather than the primary use case for OpenNLP, despite much - if not all - of the functionality being available.

```
$> opennlp POSTagger opennlp-models/en-pos-maxent.bin < testcorpus.txt | \
opennlp ChunkerME opennlp-models/en-chunker.bin

Loading POS Tagger model ... Loading Chunker model ... done (0.313s)
done (0.604s)
[NP This_DT eBook_NN ] [VP is_VBZ ] [PP for_IN ] [NP the_DT use_NN ] [PP of_IN ] [NP anyone_NN ] [ADVP
     anywhere_RB ] [PP at_IN ] [NP no_DT cost_NN ] and_CC [PP with_IN]
[ADVP almost_RB ] [NP no_DT restrictions_NNS ] [NP whatsoever._WP ] [NP You_PRP ] [VP may_MD copy_VB ] [ADVP
     it,_RB ] [VP give_VBP ] [NP it_PRP ] [ADVP away_RB ] or_CC
```

Figure 2.3: Chaining multiple parts of the opennlp system together to build a more complete analysis; in this case POS tagging followed by chunking. Note that for line-length reasons the command has been broken out to two lines.

This separation of individual processing units is characteristic of a trend towards smaller tools being used in conjunction to build up far more complex systems, both in the NLP space and indeed most scientific fields.

2.2.2 Build and Workflow Systems

Snakemake[31] is also evidence of this general shift in design style. Generally used for bioinformatics processes, it presents a scriptable interface for running easily repeatable analysis, or as the project describes itself:

> The Snakemake workflow management system is a tool to create reproducible and scalable data analyses.
>
> - Retrieved October 2019 from https://snakemake.readthedocs.io/en/stable/

Workflows are described as a series of dependent commands, with the input and outputs of each tracked such that they can be used to determine the execution order automatically, an example of which is included here in Figure 2.4. This is similar to GNU Make[8], whereby the states and paths of input and output files are used to build up a complete dependency graph during the initial stages of the build. Traversing this graph is then used to generate a sequence of commands to actually execute on the host system.

```
rule targets:
    input:
        "plots/dataset1.pdf",
        "plots/dataset2.pdf"

rule plot:
    input:
        "raw/{dataset}.csv"
    output:
        "plots/{dataset}.pdf"
    shell:
        "somecommand {input} {output}"

```

Figure 2.4: An example of part of a Snakemake configuration, note the rules defined by the input and output commands used to build up the order-of-execution relationships. Taken from https://snakemake.readthedocs.io/en/stable/, October 2019

This same graph can, in the case of Snakemake through static analysis, determine which sections are independant of one another, and can therefore be run in parallel, producing branching paths.

Common Workflow Language (CWL)[41], LAPPS Grid[26, 27] and the Computation Flow Orchestrator (CFO)[11] all also attempt to do this for various domains, with CWL presenting the most general purpose approach, lending itself to no particular analysis toolset or research field in particular.

These languages and smaller tools are complimentary to one another, provided that the formats used are understood by each tool. Some of my own work has attempted to address this interoperability problem in CL-MetaHeaders[53], where working towards a common format, or common header to identify formats would help conforming programs communicate.

2.3 Middleware

The management of these large connected systems of processes is an inherently difficult task, and as such rather than relying on the relatively primitive structures and interfaces provided by

the underlying operating systems, many applications use 'middleware' to bridge the management gap.

As previously mentioned, there are common workflow-oriented designs such as GATE, but as a more general solution, projects such as Apache Hadoop[58], which comprises a complete stack from storage through task allocation and processing distribution is probably the most well-known.

While it is technically possible to configure a general purpose system to work as all aspects of a Hadoop system, the difficulty in doing so along with the minimal gains mean that far more frequently Hadoop systems work more like a cloud network than anything else; taking jobs from external hosts and applying them to a cluster of dedicated Hadoop machines as a whole. This does tend to push Hadoop outside the scope of general purpose computing, making it a significantly less attractive solution overall.

While Hadoop offers a heavyweight solution, lightweight messaging systems such as Message Queuing Telemetry Transport (MQTT)[15] have recently become fairly popular. In a similar manner to RPC or structured message schemes, the developer is left to build the actual software infrastructure, instead just deferring the transmission and reception of data over varied network links to the messaging layer.

A departure from middleware designs in the traditional sense, MQTT (and similar protocols) do not have a central authority dictating the dispatch of jobs, but rather have a singular location through which messages are 'published', normally as a key/value tuple - although the value need not be a singular data value, but can often be complex data[9]. Clients interested in receiving the data published can 'subscribe' via the key, usually via some kind of filtering system, and receive messages sent to the server (or *message broker*).

This is of particular interest, as the model presented to the clients is that of one which approximates a network router - the point-to-point communication done via TCP/IP (in most cases, although reliable UDP schemes do exist) and the packet store-and-forward schemes handled in the middleware 'routing' layer.

While lightweight, protocols such as this are just that - protocols. They provide the building blocks for the rest of a middleware system, but still rely on the system layers themselves to do the actual message transport, resulting in network overlay schemes.

While not directly 'bad' *per se.* having the semantics of the network in higher layers precludes the system from doing beneficial things to the network messages 'in flight' - broadcasts become multiple unicast messages, for example, requiring multiple copies of the same message to be sent as distinct entities, effectively ruining the minimalist goals of the system-level transport layer.

To fully understand the behaviour of the system, and how it can be altered to manage different interactions, it is necessary to focus next on attempts to deviate from the standard models of multiprocessing and communications found in all contemporary operating systems.

2.4 Hardware Improvements

It is worth starting with the improvements that have been made to computing hardware, as fundamentally, the design decisions made at this level will tend to mould the higher layers into similar forms.

In recent history, the number of discrete processing-elements available per host has increased dramatically, taking individual computers from single core processors to 56, or more, cores in high-end multi-die server configurations[10], and typically very capable coprocessors integrated as well. Beyond the base configurations of these hosts, discrete coprocessors such as GPUs become common, and the number of processing elements they contain increases exponentially with increasing cost.

Moore's Law predicted the doubling of transistor density approximately every two years, which corresponds to an increase in processor capability each time. The 45 nanometer (45nm) lithography process, along with the 32 nanometer process brings component size down to scales where quantum effects start to become more apparent between individual signal lines causing internal interference in the processor itself, effectively putting a limit on the possible improvements in available compute resource based on the size of the silicon and its ability to dissipate heat.

With the central processor beginning to trail off in performance at 8 cores (9 in some specialist processors), attention has drifted to GPUs, where the chips in GPUs allow the integration of processor core counts in the hundreds, if not thousands (The nVidia P100 GPU Accelerator[11] has 3584 Cuda® cores, for example).

Alongside this progressive increase in local computing power networking capabilities have also improved. Even Small Office Home Office (SoHo) routers are capable of gigabit throughput rates and real-time packet filtering, and support for IPv6 is now common on local networks.

In combination with increased processing capability and faster interconnects, there have also been incredible advances made in memory density. Low-cost computers now routinely contain multi-gigabyte RAM arrangements (often also with expansion available for more), and easily over 100 GB of long-term storage, even in those using solid state disks (SSDs).

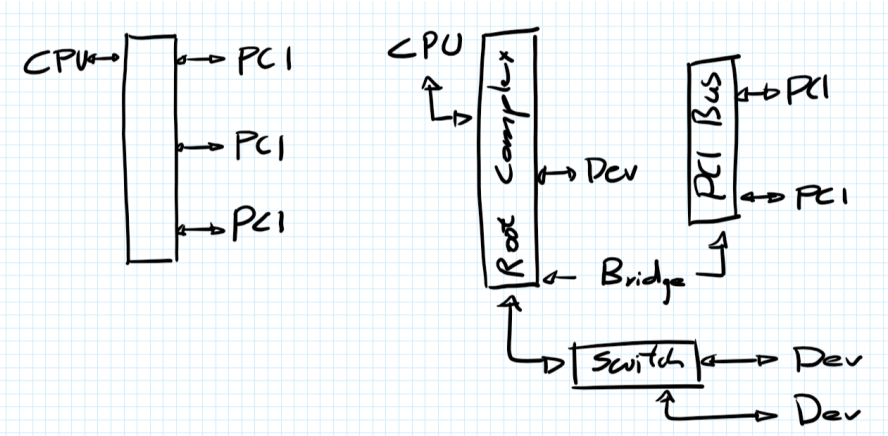


Figure 2.5: The PCI bus verses the PCIe network

2.5 Changes in the Platform

PC interconnects appear to be progressing along the same technology sequence as data telecoms networks have already traversed. Pre-PCI and traditional PCI/PCI-X systems used bus network configurations, relying on the receiver electronics to determine if the data in question was intended for that device - effectively a broadcast-always system.

Beyond PCI/PCI-X, with the creation of PCIe, the topology has been radically altered to a circuit switched network of bidirectional pairs transmitting serial data. Routing in this new environment is achieved by intermediate *switches* selecting which connection to forward via as the data passes through them. This moves the intelligence off the individual devices and on to the motherboard and its controllers.

In Figure 2.5 the two architectures are represented side-by-side, and the changes can be seen. The new architecture, as shown on the right, bears striking resemblance to an Ethernet switched network, with switch fabric between the devices to enable momentary direct, physical connections along specific paths, eliminating collision issues inherent with bus-type connection schemes.

In this form, connections between transmitter/receiver pairs are made by physical connections allowing wire-speed transfer beyond the initial route-forming transmission. The limiting factor becomes the upper bound transfer rate of the communication chips in the devices, much like the dial-up telephone communication network, where throughput was (and indeed, still is) limited by the clients, rather than the network itself.

One important feature of PCIe should not be overlooked; devices are not limited to a single network channel. 1, 2, 4, 8 or 16 channels are supported by the physical layers currently available, but the actual upper bound on this number is dictated by the hardware vendors, rather than the specification itself. Indeed, in the cases where multiple devices are further connected out-of-band with PCIe, such as the case for high-performance video cards, 32 concurrent

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Byte +0	R1	Format			Type				R2	TC			R3			
Byte +2	TD	EP	Attr			R4		Length								
Byte +4	Requester ID															
Byte +6	Tag (Unused)								Last BE				First BE			
Byte +8	Address ...															
Byte +10	... Address cont.d														R5	
Byte +12	Data DW 0 ...															
Byte +14	... Data DW 0 cont.d															

Figure 2.6: The data structure of a PCIe 'network' packet

physical connections are possible (More in the case of *Crossfire* boards by ATI/AMD, which with 3 cards could reach 48 serial connections).

The move to this network arrangement has also meant that the data flow behaves in a much more TCP-like manner. Connections have a life cycle; setup, transfer and tear-down stages all have overheads associated. In the setup phase, the physical route to the destination is constructed. Switches along the path build links attempting to avoid congested paths in a very MPLS-like manner. With a path set up, the data can be freely transferred between the endpoints without fear of congestion, contention or collision, and can be likened to the established session phase of TCP.

The patterns of serial links between endpoints is repeated throughout the modern system to span networks beyond the single host, and further out yet to organisational layers around host-groupings.

Following this technology progression we might expect the next stage to be a move from *circuit* switched networking to *packet* switched networking. However, a move like this would require significant performance improvements for the network switches with very little actual gain in terms of overall system performance - assuming a system in a similar form to those we see today - something which is unlikely to be made, as manufacturers will be resistant to this kind of change.

The similarities with Ethernet and other switched network architectures are continued to the data transmitted, with PCIe messages formatted in a very similar way to macro-network[12] messages.

The basic frame structure is shown in Figure 2.6, but is extended by additional data as required by the specific task in operation, exactly how higher-level network protocols, such as TCP, extend their lower-level counterparts, such as IP (and indeed IP to Ethernet frames).

2.5.1 Zero-copy Processing

Fundamentally, copying memory around a single system is an expensive task, both in time and (storage) space, and the costs inherent with performing a copy scale with the increased packet sizes and counts, amplifying the problem.

This has lead to a number of efforts to tackle anywhere in the processing or forwarding

of messages where a copy would traditionally happen through 'zero-copy' techniques instead. These techniques either use careful marshalling of data or memory tricks (such as memory space mapping) to process messages in-place without ever having to copy them beyond their initial ingress to the system.

Two particular frameworks which attempt to do this for network traffic are the Data Plane Development Kit (DPDK)[29] and Netmap[46]. Both attempt to provide a mechanism for userspace processes to access message buffers more-or-less directly, without having to have the system copy the messages into userspace first, and are gaining support for cases where much of the data processing is to be done by discrete processes in userspace rather than just delivered to a fixed endpoint.

2.6 Asymmetric Multiprocessing

While the challenge of managing a number of identical processors (or functionally equivalent ones, ie. all x86 or x86_64) is complex in and of itself, when the processors have varying capabilities this becomes an order of magnitude more difficult again.

GPUs, FPGAs and other specialised processing elements offer much greater efficiency for specific tasks, but suffer when given others. This efficiency gap presents two problems for systems attempting to use these processors:

Quantification Simply being able to put a meaningful value on a processors' ability to handle a given load is difficult. Capability systems design offers a possible solution, but do require prior information on a given device before it's metrics can be used. (But this in itself is no worse than drivers in existing systems)

Execution In most cases, the architecture of the "exotic" processor is different to that of the executing peer. This presents a compilation and execution problem, whereby all code must be compiled for all available architectures, or must be in a form that can be made to execute on all architectures (through the use of JIT, LLVM, VMs, etc.)

FatELF formats solve some of this for common platforms by bundling the ELF sections for several architectures together in a single binary.

JIT compilation is another good approach for programs executing in unknown platforms, but then has the overhead associated with compilation-on-demand.

A good hybrid of FatELF and JIT may be LLVM compiled code, where the final binary is generated on the host during installation time, keeping the distributed "binary" in LLVM form.

This may introduce strange transient errors due to differences in the compiler for specific platforms. Even compiler versions or optimisation levels may have an effect on the final program behaviour in particularly sensitive applications.

2.7 Networking

Moving beyond a single machine, the network characteristics and how they are handled by the operating system also come in to play. Originally nothing more than simple serial interfaces between machines, we have seen an explosion of complexity and capability since the first designs for networks.

Even simple, low-end networks are now highly capable in their own right; largely due to the standardisation of interface mechanisms between machines, allowing enormous interoperability. It is now expected that one may take any given network-capable device, and attach it to any local area network, often wirelessly, and be able to connect to other hosts with reasonable reliability (network rules permitting).

While still orders of magnitude slower than the internal bus lines on a given host (PCIe 3.0 is capable of transfer speeds of ≈32GB per second), commodity local networks are easily capable of handling impressive speeds; 1000BASE-X networks peak at ≈125 MB per second, and high-end data centre networks running 100GBASE-X networks peak at ≈12.5 GB per second.

Infiniband[13] achieves a throughput in excess of current PCIe 3.0 technology, with Infiniband EDR 12 peaking at an impressive ≈37.5 GB per second, and is expected to be double this in 2017 with the introduction of the HDRx specification[14] easily bringing the effective throughput, including protocol overhead in-line with the internal bus speeds of hosts.

Naturally, this is lower than the internal processor interconnect speeds used in the hosts, which can, for example achieve 9.6 GT per second (Intel QuickPath Interconnect) between processor cores, and 102 GB per second to memory[15] - but with this kind of throughput the differences in remote access when compared to local access become reduced.

2.7.1 Network Representation

One of the largest factors in determining how an operating system design approaches managing its processes comes from how the network is represented to the rest of the system, and conversely, how the system is presented to the rest of the network. By placing barriers along the boundaries that networks lie, the overall effect is that the internals of the system are to be cut off from the rest of the network. Alternatively, by not having these barriers, the system is much more likely to be presented as a kind of group compute resource, allowing networked applications to span over multiple systems.

These two behaviour patterns are entirely opposed, and present characteristics that shape the rest of the function of the operating system.

2.7.2 Hybrids and Overlays

Spanning the gap between isolated systems and shared systems are hybrids of the two, where some portion of the system is exposed to the wider network, but the system itself runs independantly of the rest. This is the case in Apache Hadoop, for example, where job runtimes are dispatched to the worker nodes to be executed, along with data, if required, but the actual nature of the system isn't known to the service whatsoever - exactly how those worker nodes actually allocate the space, time and connectivity to be able to execute those jobs is not known.

Other hybrid approaches include Docker, which provides limited-scope runtime overlays, allowing processes to *apparently* run in relative isolation (They still can access the rest of the running system, such as *ioctl* files and system calls), but otherwise present a pseudo-virtual machine to the rest of the network (assuming the docker worker is a network job, of course).

2.7.3 Flow Marshalling

Recently, a number of technologies have emerged for handling network traffic in a number of interesting ways.

Obvious bulk-effort attempts to manage network flows have been around for decades, with approaches such as random back-off, various funneling techniques such as RED Queueand associated techniques for flow control quickly becoming exceedingly common. Even the lowest-grade software-only router can now easily handle the required throughput or queues to be effective without harming the network flows for example.

However, with all the advances in this field, the ability to *identify* and *track* individual *flows* has become key

2.8 Scaling and Migration

In parallel with the work taking place to create smaller, modular systems and be able to easily spin up sand-boxed virtual machines there have also been efforts to create ecosystems that support this new service model.

Amazon Web Services (AWS)[16] is a prime example of this, offering tailored specific virtual machine instances for particular tasks. By constraining developers to build services with particular structures and tasks, AWS applications can be replicated and migrated to scale with demand.

Naturally, this has significant financial advantages, as charging by usage can be metered down to much smaller units, but also shows the advantages in creating systems whereby the developer is not free to use general purpose hardware for any task, but must declare the requirements of the process beforehand. This in itself lends weight to the argument for capability systems, where this type of declarative runtime is required.

Alternatives to this declarative style use techniques such as static analysis to determine the requirements of an application before runtime, but can lead to unexpected results, often requesting higher specification containers than the processes actually need.

2.9 Security

The security of systems connected to one another has become increasingly pressing, with dedicated attack groups reportedly being set up to breach specific systems. This in turn has lead to a focus on separation and *sandboxing* of applications and services in an effort to limit the range of attacks that can be performed on a given system.

Tightly controlled interactions between each *sandbox* allows the services and applications to intercommunicate, and the main way that this is done is via sockets and other network-layer interfaces, leaving the controlling operating system to route the messages either locally (as IPC) or over the network.

It is worth noting that the operating system is doing no small amount of multiplexing at the network interface layer, with an interface presenting itself as a number of endpoints simultaneously. Each virtual interface to the network card acts as a separate, complete network stack, and as many protocols commonly in use today require complex state tracking (such as TCP[28]) the stacks themselves are managed entirely separately.

This goes completely against the previous approaches to networked services, where previously there was a strong push to move the functions of the network stack out to the network card itself; many cards became fully-capable computers in their own right[17] capable of running full operating systems themselves inside the host machine; other cards embed entire Field-Programmable Gate Array (FPGA)[18] chips in an effort to move processing tasks directly to dedicated or semi-dedicated hardware for traditional processing tasks.

2.10 Small Devices and IoT Networks

At the other end of the scale, far removed from large capable servers are the ubiquitous small devices that now pervade our lives. These small devices, while generally low power and fairly quiet (in network traffic terms) can collectively through sheer volume generate vast amounts of network traffic.

In many cases, the devices in question are more like device drivers for specific hardware in a larger machine than they are a host in their own right. By treating them as such, we encounter a vast, connected computing device, capable of spanning large physical areas as well as large virtual, networked space.

From roughly mid-2015 there has been a huge increase in the number of small devices using fairly heavyweight protocols with minimal designs; heavyweight by requiring some considerable overhead to do fairly simple tasks, but lightweight by keeping the actual payload data very small and simple.

2.11 Management

Once the number of concurrently executing processes reaches a certain point, the problem, both for the programmer, and the system itself becomes one of *management*.

To further compound this problem, the behaviour of the underlying operating system and hardware becomes a significant factor. Operating systems have a number of specific quirks that cause performance drops, as policies decided in the system software determine how processes interact.

Additionally, the hardware in use will have a great affect on the way processes behave; once the hardware landscape becomes sufficiently diverse strange errors can occur, and be exceedingly difficult to debug.

As has already been mentioned in this chapter, there was a push for more capable network cards in high performance applications (and indeed, many still run today) with the cards able to run an entire operating system of their own. In this environment, it becomes incredibly complex to manage the separation of individual network stacks for applications, leading to a large number of hand-over operations between the host operating system, it's applications (and sandboxes), the card operating system, and any of the processes running therein.

The popularity of systems designed for large scale computing tasks such as Apache Hadoop[19] demonstrates the utility of the *map / reduce* approach to tackling large data sets.

By splitting a large task into individual, independent *work units*, then *mapping* them over a large number of processors, followed by a final *reduction* stage to recombine the resultant data, it is possible to vastly decrease the processing time required through parallelisation.

2.12 Kernel Design

With the progression in hardware, there was (and continues to be) a pressing need to revisit the design of the operating system. In its most basic form, an operating system kernel:

- Presents the hardware available to processes in a consistent, reliable way with the complexities of the actual hardware abstracted away (The degree of abstraction is defined by the nature of that particular kernel)
- Divides the computing resources up into time and space allocations for each process running under its control.

These characteristics can be achieved in any number of unique ways, with some having specific advantages for particular applications - Real Time Operating Systems (RTOSs) are one example of this, with specific processes provided run-time guarantees to ensure performance and timeliness of operation, and are frequently used where the applications running are safety (or cost) critical.

By far the most common design philosophy for kernel design is, however, that of fairness; dividing the resources into equal chunks to allow any process the opportunity to use its slice of the overall machine. This can be seen especially in commercial instances of virtual machine hosts. In these cases it is often paramount that the hypervisor or kernel be able to manage the demands of several sub-kernels (virtual machines) on what they believe to be an entire machine.

The following few designs illustrate some of the variety of approaches to these problems that researchers have attempted, highlighted to show the diaspora of designs available.

2.12.1 The 'Tesselation' Design

In their 'Tesselation[36]' operating system, Liu *et. al* also note that the current model for kernel design has not been designed to be used with modern hardware and that the number of cores that are being employed in modern systems is increasing dramatically. This they believe should be taken seriously as a design issue, and made the crux of the design, in addition to a constraint.

As the number of cores approaches the number of threads normally running on a desktop machine, the issues of context switching become less and less apparent, and the tear down mechanisms for caches and shared memory methods become more and more important as a factor for slowing down the execution of programs. Rather than view the system as a single core running multiple threads, the Tesselation system breaks down the hardware into 'space-time partitions' which allow resources to be allocated to threads running on separate cores with minimal impact on the life cycle of other threads.

Liu *et. al* define *Space-time partitioning* as:

> Space-Time Partitioning in 'Tesselation', Rose Liu, *et. al* ... an isolated unit containing a subset of physical machine resources such as cores, cache, memory, guaranteed fractions of memory or network bandwidth, and energy budget.

This is quite close to the definition of a virtualized machine in a normal virtual server, and has no provision for threads or processes within the 'partitions', leaving the management of multiprocessing up the partition's resident code.

This leads to a simpler kernel design, as the task scheduling can be made much more coarse-grained, and the partitions' resident code can be given direct control over the behaviour of threads in its working set.

Using this design, applications can then be written much as small micro-kernels on a resource constrained machine; an approach familiar to older games console designs, whereby loading software into memory gives the program full control over the machine. In contrast to games consoles, in which the loading software is much closer to a boot loader than a full operating system, being entirely replaced by the application; Tesselation continues to execute in parallel with the applications.

The Tesselation operating system can give hard assurances that given partitions will have certain resources available to them at all times, thus the application can completely rely on the availability of said resources, and utilise them fully without fear of them being taken by another process and have to be paged in or out of memory. This would allow, as an example, a network subsystem in a partitioned system to be given precisely as much CPU time and memory to allow it to perform at peak performance, allowing it to make latency and throughput guarantees ahead of execution time.

Furthermore, because the access to resources is being monitored by the kernel, the transactions could have traditional QoS rules applied to them, provided that the services running in the partitions can cope with the delay. This may lead to fairer scheduling (first come first serve, prioritised traffic, fixed delays, and so forth), giving the application developer a much more fine grained control over how and when messages should be run, and how much of the machine (or indeed, *machines*) a partition should be allowed to use.

In addition to this, the separation of partitions may allow a 'fast-restart' mechanism to be implemented for critical services. To illustrate the advantages of this, in the event of a network service crash, the state of its memory could be retained, then the operating system can in-place restart the network service in a partition with the same memory mapping in an attempt to resume without interruption. Of course this brings the obvious dangers of continual crashing due to memory corruption, but this could be detected with a simple maximum retry count for any given service (yet another parameter to include in the partition definition).[20]

With partitions effectively defining a set of policies for resource access for a given program or set of programs,the need for some sort of resource-permission manager is required, along with the permissions stored in kernel controlled memory somewhere. This could quickly become problematic, as the additional information takes up space for small partitions, forcing program-

mers to combine their application functionality into single, larger partitions to avoid wasting run-time storage on permissions. Furthermore, this resource management may slow down the access to said resources, as transactions between the partition programs and kernel code would have to be vetted for security before any access could be granted.

As previously stated in 2.12.1, there exists a possibility of using QoS-style rules for a partition, providing a mechanism for well-defined guarantees for performance and resource availability, unfortunately, this adds overhead to both run-time and partition creation events. Checking the QoS rules for the system and the requested partition settings may cause considerable latency in creating a partition (in a busy system), and during run-time, any dependency on resource availability could become a problem as partitions are created and destroyed around each other.

In some cases, applications may wish to use multiple partitions to gain additional security, such as a web-service using a separate partition for script interpreters allowing them a safe sandbox-like environment to run unsafe code in. However, this presents its own difficulties, as programs would have to be written to cope with fast-restart functions in components of their operation, and/or several partitions would have to be halted during a fast-restart to prevent data access on a 'dead' partition.

Tessellation uses a kernel-level message passing mechanism to facilitate IPC, bringing both the advantages and disadvantages inherent in a message-based system with it. (Namely, latency, asynchronous calls, QoS, kernel-level buffers, and so forth) As an alternative, Rose Liu *et al.* do talk about implementing a mechanism for limited shared memory between partitions, although this would appear to break their security model by punching a hole in the separation of partitions.

While the work undertaken in Tesselation has made significant inroads into producing a better model for an operating system, it is the opinion of this author that work still remains with optimisation and implementation and the challenges inherent therein. Additionally, the work omits exactly how multi-core systems should implement this model, merely define a structure for how applications may be contained in a multi-core environment.

2.12.2 The 'Exokernel' Design

The 'Exokernel'[17] design requires a very small kernel which has the sole purpose of exposing the hardware in a way that allows more generic operating system 'libraries' to actually perform that which would normally be the kernel functions. This model requires that the kernel itself is as simple as possible, whilst also effectively multiplexing and marshalling the various requests for hardware access.

By keeping the complexity of the primitives exposed by the kernel to a minimum, the effi-

ciency of the operations required to produce them can be maintained, keeping the processing time down. Furthermore, with only very minimal abstraction being performed at this level, the flexibility of the overlaying libraries is not compromised by unfolding already abstracted interfaces to hardware.

This allows the libraries to provide a myriad of different interfaces to hardware, as there is no obstruction to abstraction. Once the barriers of pre-existing primitives have been removed, the libraries are free to abstract the functions of the operating system in any way that the developer sees fit, including extending or replacing fundamental operating system operations.

With the majority of the operating system implemented in libraries, the number of kernel-user space boundary crossings is reduced, as the processor spends most of it's time in the user-space libraries, only transitioning to perform actual hardware access, such as disk IO, or communications.

As the kernel only provides the minimal required primitives for accessing hardware functions, the access barriers for processes are smaller, allowing for near-direct access to hardware from user-level services.

This brings a level of performance computing unachievable through the use of virtual machines to create the same abstractions, simply due to the program's proximity to the hardware (as illustrated in Figure 2.7) , with the abstraction and re-abstraction layers removed.

The kernel itself makes little or no assumptions about the purpose of accessing a hardware resource, leaving the management of the resource itself up to the libraries. All the kernel does, is to ensure that each process with access to hardware does so without disrupting any other processes access, separating the protection of processes (which is handled in user-space) from the management (multiplexing) of services.

The structure of the Exokernel has a few problems with the way that standard x86 or x86-64 architecture computers are constructed. One of which presents itself during boot, where the kernel (which only knows of the notion of a disk controller by its address) is unable to load any of the system libraries to perform bootstrap operations.

2.12.3 Implementation - Barrelfish

Many current multiprocessing frameworks assume that all processors are created equally; unfortunately, this simply isn't true in a large number of cases. Excluding computer systems that have been designed and built with the express intent of being used for multiprocessor work, the modern computer is constructed with a myriad of different processors, each with their own particular niche of high-performance.

The 'Barrelfish'[5] kernel presents a very small requirements set with the express intent to be

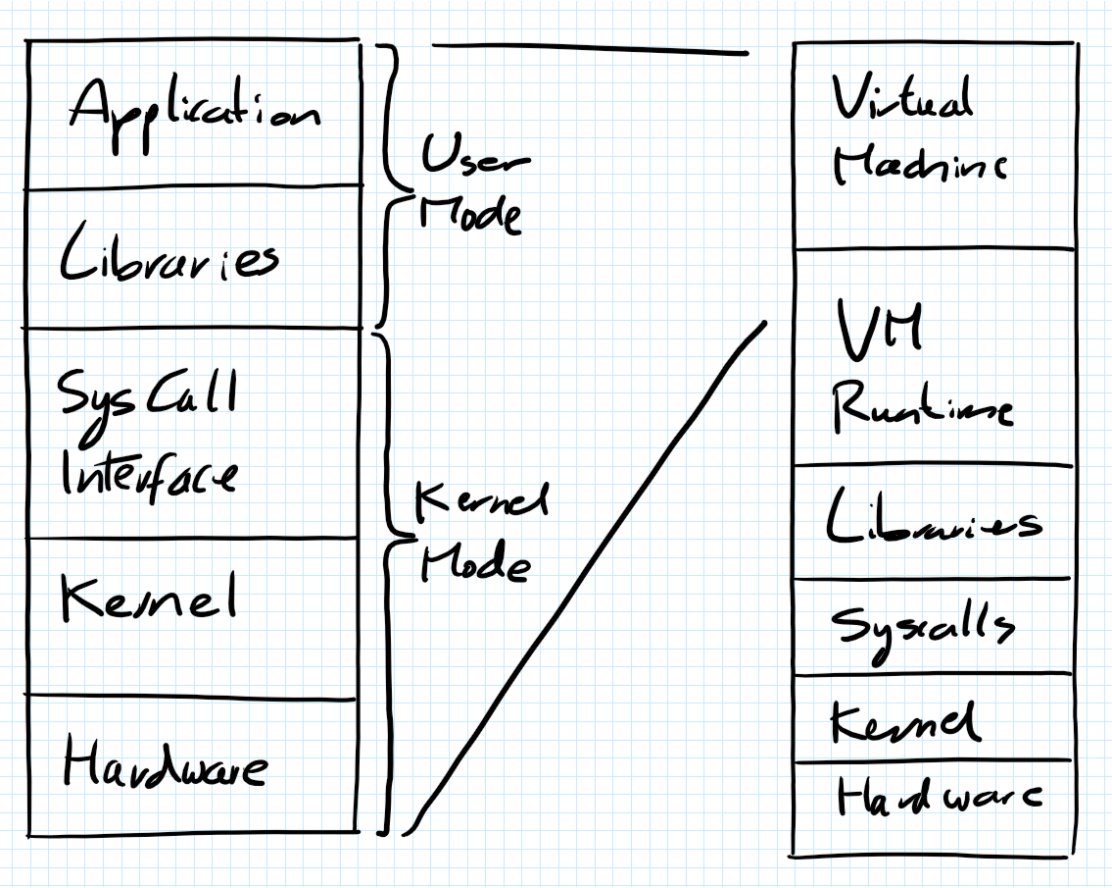


Figure 2.7: An approximation of the layers between normal application code and the hardware they are executing on in traditional and exokernel model operating systems

able to be executed on a large number of processors. This allows kernels to be deployed to programmable devices attached to the main machine, such as graphics cards and programmable network interface cards.

Once deployed to a given processor, the kernel assesses the hardware available to it and loads up any required drivers to provide minimal access, then waits for a coordinator kernel to assign a process to it. While this reduces the complexity of processor look-up, it does so at the expense of scalability as each processor in a machine needs to be able to communicate with that single one to be part of the system.

Throughout its development, the Barrelfish operating system was designed with the following requirements in mind;

- To *at least* match the performance of existing traditional operating systems.
- To easily and efficiently scale across large numbers of *diverse* processors.
- Allow easy migration to new platforms by keeping the code changes minimal.

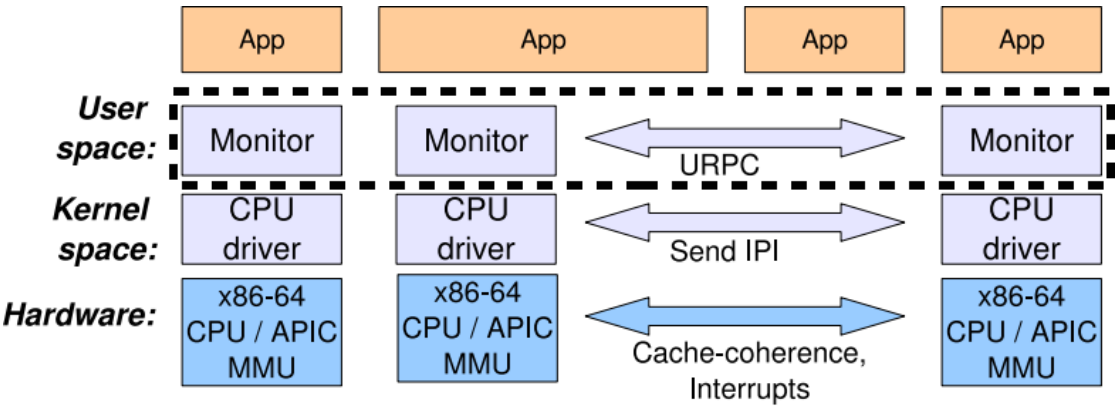


Figure 2.8: The Barrelfish architecture, as presented in"*The Multikernel: A New OS Architecture for Scalable Multicore Systems*"[5]

- Use the message passing mechanism for pipelining and batching to increase performance.
- To allow what would be traditional OS services to be migrated close to related hardware.

Rather than packaging the traditional kernel functions such as thread support, memory management (allocation/de-allocation), and memory replication as kernel libraries - as would be the case in a traditional operating system - these functions are performed in *Monitors* (see Figure 2.8), processes that are associated with a specific processor which sit between the kernel and any user-space applications running on the Barrelfish system.

Although the *monitor processes* allocate and free memory for user-space processes, they are not hardware aware, and can allocate more memory than the current processor has available to it. In this case, the message passing IPC mechanisms kick in and allow memory to be seamlessly allocated in other resource pools.

For shared memory or memory copy operations, Barrelfish uses multicast-style messages to propagate changes in state as well as any IPC required. This is achieved by means of a common interface to one or more methods of communication, which actually perform the transfer - providing the application with a transparent pipe through which data can be exchanged.

This works across domains, such that a process communicating with another on the same machine over the PCI/PCIe bus uses the same mechanisms as one communicating with an entirely separate machine over a network interface. By doing this, processes can be migrated away from each other - even across machines, provided there is a network link between them to facilitate the communication channels.

The messaging system extends to the mechanism for state replication, which messages marshalling memory at remote locations to keep relevant memory up-to-date across the system. This means that state change is portable across different transmission methods, allowing for a

common memory management system in all kernels.

This does lead to a system where message queue marshalling is rather critical to the operating system as a whole, as all process intercommunication and memory management is done through the same channels. It was quickly apparent that any changes in how the OS managed the message queues (or differences in how the CPUs managed the queues) added up to curious anomalies in performance.

Barrelfish appears to be a step in the right direction, in terms of architecture at least. By their own admission the solution is not optimal, and could easily be improved upon, and the array of hardware tests that were performed was rather limited, causing their own performance benchmarks to be somewhat unrepresentative of what real hardware would do in 'the wild'.

2.12.4 Implementation - Corey

In contrast to Tesselation and Barrelfish, Corey[8] seems much more concerned with how data is used in the system over how the system integrates into a larger computing entity. This is useful, as the authors (Silas Boyd-Wickizer *et al.*) have spent considerable time on examining the viability of using the POSIX standard in a more multi-core aware way.

Noting that the number of cores in a given processor has been increasing over the years, and that the total memory available to the processor has not, the Corey group point out that the amount of memory available to each individual core is actually getting smaller (if we only consider immediately available, high-speed memory stores, such as RAM, rather than long-term storage).

While the cores do have access to other each other's cache, the paths through which this memory can be accessed are far slower than the path to the memory allocated to each core. The read/write lag gets even worse if several cores have to be involved with the operation through routing.

Furthermore, the way that processors are built is having an impact on how the memory access paths are arranged, such that there are (usually) internal groupings between cores, giving a very uneven memory latency for memory access as can be seen in Figure 2.9.

This routing for memory access has a further knock-on effect; the processors involved in the routing usually have to stop execution of their own (possibly) unrelated tasks to act to move memory around the processor, such that knowing the physical layout of the processor becomes very important for memory arrangement.

Although this paper does not mention the concept of a 'processor driver' it does match very well with what they appear to want to achieve, as this would allow the various mappings for each processor core to be organised in a way that would allow them to be used with the rest of the

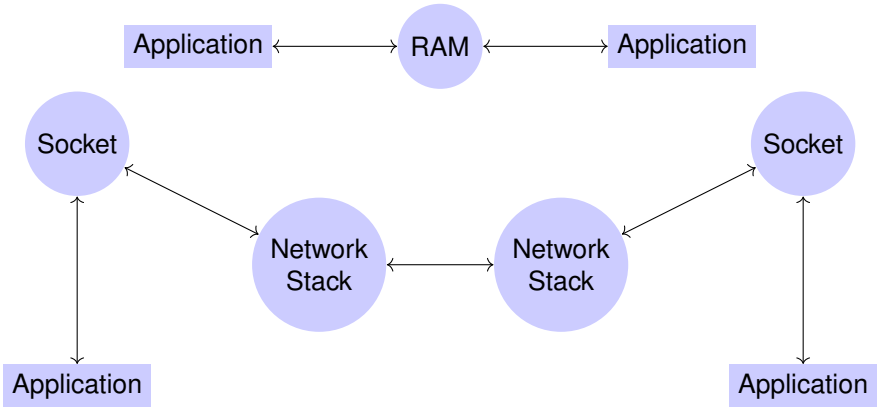


Figure 2.9: Routes through hardware communication paths, demonstrating the variety of latencies paths exhibit

operating system with minimal (if any) impact on the rest of the system.

The 'Corey' operating system uses the Exokernel design (See 2.12.2)

2.12.5 Implementation - Fabric

The Fabric[35] system is designed to provide a secure platform for distributed computation, and achieves this through trust relationships between *Nodes* in a network and a modified Java environment.

Programs are written in a Java derivative (Fabric) and its intermediate language (FabIL) which combine Java Information Flows (JIF)[21] and some adapted GNU Classpath libraries.

'Fabric' is a high-level programming tool, on roughly the same level as the Amazon Cloud and with approximately the same goals.

The system can be broken down in to various 'Nodes' as illustrated in Figure 2.10;

- 'Storage Nodes' hold and maintain persistent data
- 'Worker Nodes' perform computation
- 'Dissemination Nodes'...
  - ...copy objects (data and code)
  - ...increase availability (data and code)

These nodes connect to storage directly, and can both save and load data. They do not perform any calculations on the data objects, but instead hand them off to Worker or Dissemination nodes for processing and distribution.

The only processing a Storage Node performs is to ensure that the data being *written* is from a trusted source, and that it is newer than the object it currently has in store (to prevent

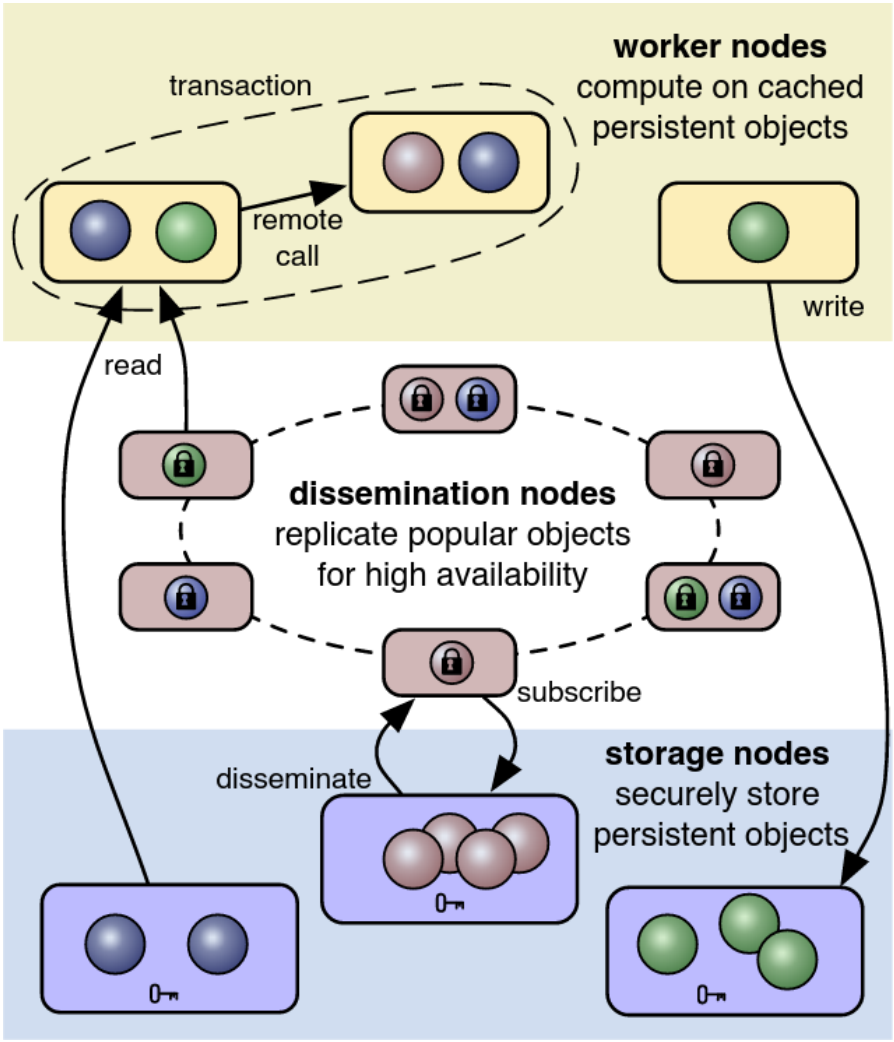


Figure 2.10: The Fabric system overview, as presented in "*Fabric: a platform for secure distributed computation and storage*" [35]

stale data writes). It is possible to edit data directly on a Storage Node (as is described in the next section) but this will only occur when large amounts of data are having particularly simple operations done to them.

Worker Nodes actually perform any computation on data in the Fabric, and will only do so if the programs they have been provided with are from trusted sources. Furthermore, Worker Nodes will only run programs written in the Fabric's own language (FabIL) although from there they can process other languages during runtime provided that the node the program is running on supports said other language.

Worker nodes operate in one of two modes during computation; they can either be running code locally and calling data objects from Storage Nodes and Dissemination Nodes, or they can load all the data required to run, then process the data using a form of RPC to execute program data on the node holding the data.

Exactly which mode of operation is used depends on the data/code cost ratio, whereby if it is more 'expensive' to copy the data to the node, it will use RPCs, and leave the data in place, whereas if the cost to transfer the program to other nodes is more 'expensive' the Worker will copy all the required data objects to itself to perform computation on them in place.

The function of these nodes is quite simple - to bring the data closer to the objects that require it.

This is achieved by copying data from the Storage Nodes and propagating it throughout the system to increase availability and reduce the 'costs' described in the previous section.

Interestingly, there is no concrete specification on exactly how data should be transferred between Dissemination Nodes - leaving them open to use any transmission method they can. While this may appear to be a security hole, in reality it is not, as the Worker and Storage Nodes will only use data from others they have formed a trust relationship with (this includes all nodes back through a given data path), preventing any malicious node from providing bad data.

Because any nodes that either store or modify any data objects must have a trust relationship that is at least as trusting as all other links in their data path, it prevents rogue nodes from changing the data en-route to a store, as the store would not trust said node.

2.12.6 Implementation - Helios

The Helios[39] OS is designed to allow platforms with heterogeneous processing hardware to better utilise the diverse processors available. This is achieved through 'Satellite Kernels' which are spawned from a single initial kernel and deployed to any and all processing hardware in the system. (PIO Cards, Network cards, etc.)

Rather than think of the kernel as a collection of drivers for hardware, the Helios paper takes the view that the kernel should be treated as a CPU driver, in the same way that any other driver in a system. This means that there should be a common interface to access a 'processing unit' which allows programs to run on top, using features either provided directly in hardware by whichever CPU the process is running on, or as software emulated hardware. This is a flip in the normal abstraction model, whereby instead of a kernel specifying a method of communicating with hardware, to which drivers adapt how the hardware *actually* works.

Because a kernel can be deployed to any number of fundamentally different processing units, the architecture is probed by the first kernel to boot then it loads up the required satellite kernels for the processing hardware available on the machine. These satellite kernels are then written to the relevant hardware's memory, and have their processors load the new program data.

Once loaded, these kernels then connect back to the kernel that spawned them and any

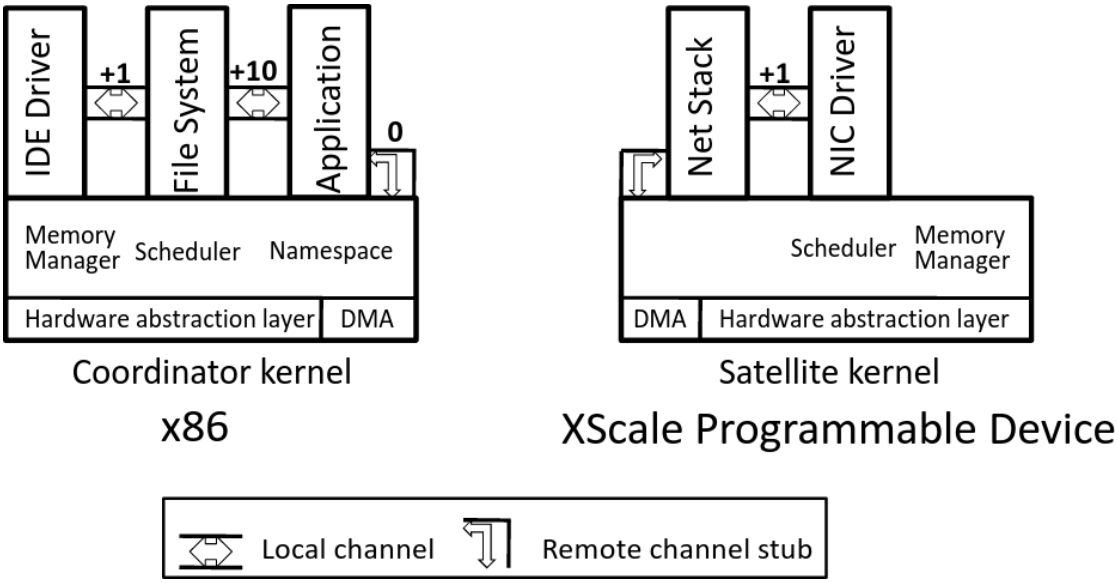


Figure 2.11: "*This figure shows a general overview of the architecture of the Helios operating system executing on a machine with one general purpose CPU and a single programmable device ... Applications on different kernels communicate via remote message-passing channels, which transparently marshal and send messages between satellite kernels. ...*" - From "*Helios: Heterogeneous Multiprocessing with Satellite Kernels*"[39]

other satellite kernels, establishing communication channels to each one in turn.

It is worth noting that each kernel maintaining connections to every other kernel works well in the systems that were tested in the paper, but only a small number of kernels were actually deployed to a system in any of the tests. They mention that this could cause scalability problems as the number of kernels increases it will begin to significantly affect the amount of memory the connections use up on lower-specification processors. It is expected that in the future, some form of routing may be required to cope with the load.

The interface to IPC is well defined in the specification the paper provides, and hides the transmission method from the processes that call it, allowing the kernel to marshal the messages along any number of communication channels - weather that be local or remote.

In the case of a remote IPC message, the kernel is free (assuming that it has a network connection available to it) to use the normal IP network protocols (TCP/UDP) to communicate, otherwise local messages are far more likely to be sent along interconnect busses in a given machine (the specific nature of such channels depends on the nature of the machine).

All of this combines to a transparent method of communicating between processes and kernels, allowing processes to be processor independent, and thus be called from any combination of processing units.

To ensure that performance is not lost spending time transmitting messages around the

system, the kernel is designed to keep both local and remote communication to a minimum, and only to perform IPC when absolutely required.

Namespaces are used to arrange services into a manageable structure in a Plan9[38] style manner, and any non-programmable IO card is represented by a single endpoint connected to a driver on the nearest kernel to the hardware.

The namespaces are managed by the coordinator kernel, which proved to be sufficient in the small-scale tests that were performed in this paper, but it was noted that this could quickly become a bottleneck as the number of satellite kernels increases possibly requiring a distributed service discovery protocol of some sort being integrated into all the kernels.

During the tests performed on this architecture, notable increases in performance were seen when dealing with large numbers of concurrent requests when put against a more traditional OS. This can be clearly seen in their PNG file decompression tests, which showed an overall approximate increase of 110% speed-up.

2.13 Summary

As has been discussed in this chapter, the natures of both the hardware and workflow support that is in use today are much removed from that of as recently as 10 years ago, and because of this some radical rethinking of how both operating systems and their interconnections - both in software and in hardware - handle the hardware is required.

Furthermore, the hardware used to connect these systems together has improved greatly, and the very nature of the network has changed. A much tighter service integration is happening, with local applications routinely using remote services as if they were on the local host machine, rather than running them resident, and the communications between these applications and their services, and indeed, all the layers in between have become exceedingly important.

While various approaches to create a new model for the operating system have been mooted, none has currently taken hold in mainstream deployments, but instead the ideas used in several have been ported to more traditional systems, such as Linux, to present similar interfaces to those in the prototypes.

With this in mind, along with the increased complexity of tooling as described in the opening sections of this chapter, demonstrates a need for a new interaction method that incorporates some of the middleware features available now, but implements them in a way that enabled the system itself to perform the required operations more effectively than the current offerings.

To this end, the next chapter will discuss their integrated whole with the goal of producing a singular idealised model for the communication and marshalling of messages in a way compat-

ible with the workflows and models discussed here, along with a discussion as to the behaviour along the ‘edges’ of processes as they communicate with each other.

Footnotes

[4]GATE does in theory allow for true parallelism, provided that dependencies are distinct and separate - although GATE itself is not handling this interleaving, the underlying operating system is

[5]The most up to date, complete list can be found at https://gate.ac.uk/gate/doc/usecases.html

[6]https://opennlp.apache.org/

[7]http://www.nltk.org/

[8]https://www.gnu.org/software/make/

[9]At the time of writing, the current specification is 3.1.1, and can be found at http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html

[10]Intel®Xeon®Processor E7-8870 v3 - http://ark.intel.com/products/84682/Intel-Xeon-Processor-E7-8870-v3-45M-Cache-2_10-GHz?_ga=1.64899183.1500092289.1478029206 (2016)

[11]http://images.nvidia.com/content/tesla/pdf/nvidia-tesla-p100-PCIe-datasheet.pdf

[12]Macro-networks include any external to the individual host, including LANs, WANs, MANs, and so forth

[13]http://www.infinibandta.org/

[14]http://www.infinibandta.org/content/pages.php?pg=technology_overview

[15]Intel Xeon E7-8870 v3 - http://ark.intel.com/products/84682/Intel-Xeon-Processor-E7-8870-v3-45M-Cache-2_10-GHz

[16]https://aws.amazon.com/

[17]The O2E-100 PCI Express Cavium OCTEON II CN68XX Packet Processor Card is an example of this type of card: https://parpro.com/product/o2e-100-pci-express-cavium-octeon-ii-cn68xx-packet-processor-card/

[18]The Bittware 520n is an example of this approach: https://www.bittware.com/fpga/520n/

[19]http://hadoop.apache.org/

[20]It should be noted that this *fast-restart* mechanism is not unique, as it was implemented successfully in the CosmOS operating system.

[21]http://www.cs.cornell.edu/jif/ - Java Information Flows (2016)

Chapter 3

Analysis

As discussed at length in Chapter 2, the current models used to describe computer systems and their interactions are imperfect or incomplete - mostly as a result of the rapid progression of the technology - and there have been a number of attempts to redefine how we describe the process operating environment. Many of the approaches have drawn inspiration from the macro-scale, modelling internal interactions the same way that external host-to-host operations are performed on more traditional systems. Others have looked directly at the hardware and followed the connectivity described in the system itself, scaling up to the operating system and beyond to replicate the design at each level of granularity.

In this chapter, rather than attempting to combine the prior approaches into a single coherent whole, a 'clean room' approach is taken, whereby the features of the communication are taken in isolation and used to build a model from which a prototype can be constructed.

3.1 Automation and Flow

Naturally, when faced with a complex multi-step process, if individual operations can be divided, there is a natural design whereby stages work largely independent of one another, with the required information or data being the sole shared components.

However, this separation is not always as easily described in actuality as it is here, *data dependency* intrinsically ties process steps together, requiring tighter and tighter integration. This is one of the things that systems developers grappled with when the first multiprocessor systems became widely available; precisely how do we work with the 'other' processor?

Obviously, multiprocessing systems had existed before, most commonly in the guise of time-sharing systems of one variety or another. A single processor running otherwise disconnected processes for multiple users, necessitating the inclusion of locking structures and controls around critical data and resources in the machine. When faced with an environment such as this, it is easy to see how it would quickly become difficult to escape this mode of operation -

access control for everything makes for a clean design overall.

3.2 Connectivity

However, rather than attempting to categorise or describe these systems and work backwards towards a model which fits the best aspects of these various designs, instead taking a step back and analysing the space from a purely theoretical standpoint is preferable. This is mostly because by doing so, any bias in the prior designs to conform to particular hardware requirements or limitations can be discarded, ideally leaving us with as close a design to 'perfect' such that there can be such.

3.2.1 Linear Chains to Graphs

Given that the complexity of the computing surface has increased dramatically, and that the available representation used to describe process interactions has not, it is a fairly straightforward conclusion to reach that the overall utilisation of the available processing is going to be sub-optimal.

This can be demonstrated clearly through diagrammatic representation:

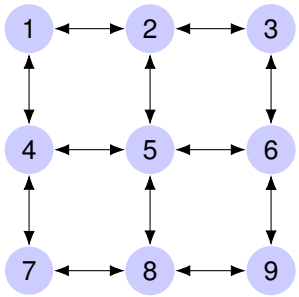


Figure 3.1: A 3x3 grid of processing elements, each connected to its neighbours with a contention-free, dedicated network bus.

In Figure 3.1, we see a grid of 9 nodes, interconnected by communication paths arranged in a grid. This kind of arrangement can be created in hardware through the use of XCore[22] modules, should one be so inclined, as they provide the required 4 IO busses, one on each side of the die, but in this case, the specifics of the implementation are not important.

For the purpose of the following discussion, it is assumed that each processor is capable of handling only a single process (or thread) at once, and is not performing any kind of preemption, such that $oneprocessor = oneprocess$. The mechanisms through which program code is loaded on to the processors is ignored, and it is assumed that the processors all begin execution at the same time, and it is further assumed that each processor has an entirely contention-free link to its neighbours (ie. each pair has exclusive access to the connecting bus) and it is able to write to multiple busses at the same time.

It is worth noting here that this exclusive access to the communication busses would only ever in reality extend as far as the core interconnects on a traditional multicore processor (eg. hyperchannel links in the Intel architectures, for example) and any links to other resources, such as main memory, for example, are over contended links, necessitating flow control or locking to ensure that transactions are not corrupted by intermingling messages.

Using this hypothetical processor arrangement, if we want to maximise the total processing ability it follows that we should attempt to use all available processors. The simplest arrangement for this would be a serpentine path through the processors, as shown in Figure 3.2.

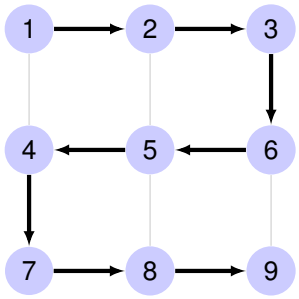


Figure 3.2: A single serpentine chain through the 9 processing elements, through which, assuming that the processing done at each unit takes equal time would result in 100% utilisation of all available processing power after 8 message writes.

In this arrangement, each processor pairing ($< 1, 2 >$, $< 2, 3 >$, etc.) has a dedicated communication channel, and can expect contention-free communication at the maximum bandwidth available at whatever clock speed they are executing on (it is assumed that all nodes run at the same speed, for simplicity).

However, it does mean that Node 2 must wait for Node 1 before it can execute, equally Node 3 for Node 2, and so forth throughout the chain. This may actually be the most optimal way of handling whatever load the application is designed to run (A 'Sieve of Eratosthenes' for prime-hunting, for example), but there are non-linear loads that may benefit better from a different arrangement.

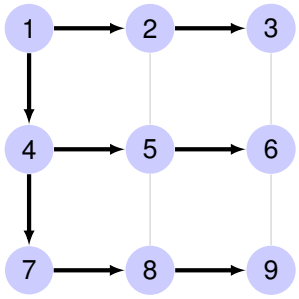


Figure 3.3: By using the left-most-vertical communication chain, this data flow could achieve 100% utilisation in only 4 message writes.

Consider the connectome 3.3. In this arrangement, the processors are arranged in a loose tree formation, whereby if the initial node is Node 1, and the workload is decoupled enough, it may be possible to distribute the overall load more effectively. Through Nodes 1,4,7 acting as distributors, data can be readily shuttled to Nodes $2, 3, 5, 6, 8, 9$.

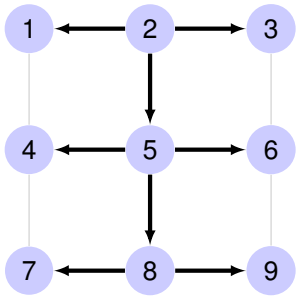


Figure 3.4: In Figure 3.3 the left-most-vertical was used to provide a faster path, but only had a maximum out-degree of 2 for any given node; in the arrangement here, using the centre-vertical path (with node 2 as the initial data source) as the main transport, the processors could achieve 100% utilisation in only 3 message writes.

However, this may also not be the most optimal use of the space, consider the connectome 3.4 instead, and it will hopefully be clear that while the same number of receiver-only Nodes is the same as the previous example, the arrangement of the distributor nodes allows for no delays in processing, at the expense of a 1-long chain at each stage of the tree. (The 1-long chain at each stage may be sub-optimal if there is tighter coupling between data elements. It is not expected that this is 'the best' arrangement, but is merely given as an example.)

All of the previously mentioned connectomes assume that data can only be initiated at an 'Edge-Node', if it could be generated, or otherwise loaded from anywhere in the graph, then the possibility of starting at Node 5 presents some interesting options.

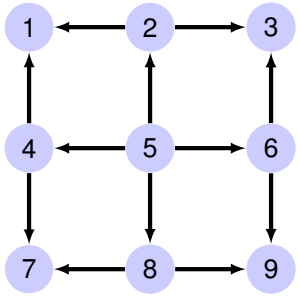


Figure 3.5: With node 5 as the initial data source, following this connection scheme, it would be possible to achieve 100% utilisation in only 2 message writes. In practical terms, if this were a real processor, getting data to node 5 for the initial processing would be problematic, as processor designs frequently only have memory interconnections around the edge, and crossing busses would cause cross-talk issues.

The connectome 3.5 would provide fewer processing Nodes than previous examples (Only

Nodes $1, 3, 7, 9$ have no out-degree), but in cases where data may require additional processing, having the option of 2-chains at any stage may prove to be more valuable than other arrangements.

3.2.2 Practicalities

So far, in this section we have been only considering the hypothetical; far removed from the actual hardware available (in 2019, at the time of writing). These models are useful, however, as they abstract away the differences between particular implementations, and leave us with a simple model on to which assertions about the nature of the processing can be applied.

Moving beyond these models, however, we become tied to the real, and need to consider the how this affects the models themselves. Luckily, as many real-world processes are input-bound (where they spend most of their run time waiting for data to process) they can often be de-scheduled, thus freeing the processor for any other waiting processes.

3.3 Technical Challenges

So far, this discussion has centered around the theoretical, but there are a number of challenges in realising this type of model. Messages in 'flight' need to be stored somewhere and be owned by either a process or the kernel itself, and the data structures needed to store these bring their own problems as well.

3.3.1 Directionality and Buffering

Pipelines of processes (Figure 3.6) can implement fairly primitive methods of flow control, as the only process that will be *starved* of data is the one immediately following the current one. In practice, this means that the actual required buffer size for connections between nodes is 0. *Zero*. In a modern Linux system, this is most analogous to the 'signal'[23] mechanism, through which software interrupts are passed directly into handling functions in the processes they are targeted at.

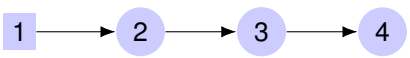


Figure 3.6: A 4-stage pipeline of processes linked by FIFO queues. The initial (square) node is assumed to be a producer, and will emit data into the chain.

Exactly what is a *practical* size for buffers, however, differs from what is *required* for buffer size. While it is entirely possible to produce a zero-buffer chain, the usual way this is implemented (and certainly the way it is implemented in Linux/POSIX[24], by default) is have a non-zero

buffer between each process, to absorb any processing time difference or scheduling oddity rather than immediately blocking on a transmission.

(a) Buffering at the transmitter (b) Buffering at the receiver

Figure 3.7: Buffering strategies for connected processes

If it is a given that a buffering strategy of some sort is required to allow processes to run largely decoupled (and thus asynchronous) from one another, the question becomes where best to place a buffer. There are two obvious arrangements; first, that where the transmitting process buffers the messages until the receiver is ready to accept them (as shown in Figure 3.7a, and second, where the receiving process buffers incoming messages until it is ready to process them (as shown in Figure 3.7b).

Both of these have subtle implementation details that can have significant effects to the overall performance of the system, both in terms of actual throughput, but also in memory usage. To illustrate this, let us consider the case where the receiver is buffering the incoming messages; in this arrangement, the transmitting process can have no information about the buffer state, so is quite capable of pushing messages until the receiving buffer is entirely full. While this may not immediately be obviously *bad* per se. it can cause the transmitter to have unpredictable run-time characteristics, as the system blocks the process from executing in a non-deterministic manner (from the perspective of the transmitter).

Flipping this configuration the other way round presents a different problem, whereby the transmitter now knows the state of the buffer, but the receiver now needs a mechanism to determine if there are messages waiting in a '*foreign*' buffer (foreign, as the memory the buffer resides in is not owned by the process).

In traditional paged-memory-sharing systems (such as Linux, for example) the system is able to perform a certain amount of trickery to make it appear that memory is present in both process' space, despite the process not actually owning the memory. Read-only page maps between processes offer an interesting way to effectively map one structure into two (or indeed, more) processes, however the the nature of the data structure in the memory suddenly becomes critical. Lock-free structures offer a method through which these problems may be solved, however, and will be discussed in Section 3.3.4.

There are ways around this problem, of course, but they involve additional support structures to notify the '*far*' end of a communication channel (from the buffer) of the state of the buffers.

3.3.2 External Buffers and System Buffers

So far, the only buffering situations described have been where the processes involved are effectively operating in isolation. Obviously, this is almost never the actual case - processes generally run on a computer system also running a large number of API and ABI frameworks behind the scenes to support the required interactions.

This leads to a third option for buffering strategy, whereby *no process has a buffer*. In this model, the *system* holds the buffer or buffers for the processes, marshalling the messages to and from the buffer to maintain the communication links.

In architectural terms, this is much more like a network router than a network switch - the internal buffer is entirely invisible to both the transmitter and the receiver, presenting itself only as a transmission delay.

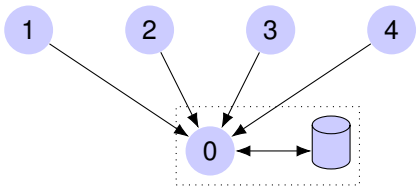


Figure 3.8: Placing the buffer in the routing process results in a much reduced memory surface, but presents more copy requirements for the router to get the messages into the connected processes.

This design also simplifies the options for allowing access to the messages in memory directly, as the routing process can allow direct, read-only access to its buffers, resulting in true zero-copy semantics and performance. However, this speed increase comes at the cost of security.

Current designs of x64 processors are only able to restrict memory access through mapping operations at the page level, sub-page mappings would quickly result in extremely large and cumbersome page tables, and indeed with RAM sizes increasing over time, the pressure to make each page larger (1miB pages and 4miB pages) is growing proportionately, making this approach on x64 architectures less likely.

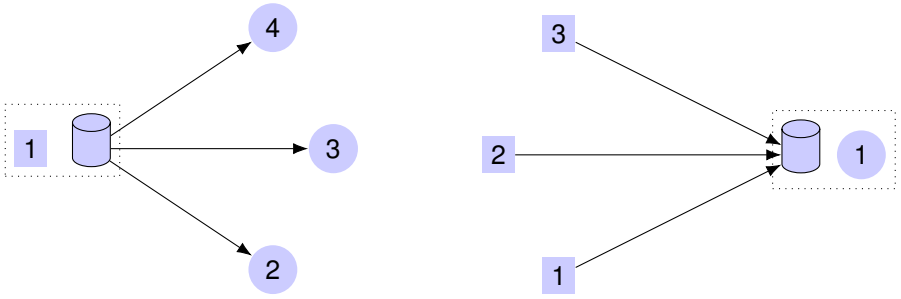
ARM processors, by contrast often have specifically engineered sections of memory which are only accessed through a message passing interface (in effect, treating their co-processors more like an external peripheral than as part of an integrated SoC[25]. As these interfaces are built in the silicon of the processor, they have special security measures placed on them that allow only selective access, avoiding the problems discussed here.

These message passing interfaces have proven to be sufficiently performant as to be able to handle extremely high bandwidth communications, for display or camera interfaces, for example, so would be possibly to use to facilitate a kind of centralised message passing store, should such

a platform be available.

3.3.3 Multipath IPC

Another particularly interesting feature of unidirectional links in this kind of structure is that they are not limited to uni*cast* links, but can instead be targeted at multiple receivers with a number of different policies to delivery.



(a) One possible *anycast* buffering strategy structures

(b) The reverse of the *anycast* model, the multi-source configuration

Figure 3.9

Take the fairly common case of an *anycast* message (see Figure 3.9a); in this case, the message is expected to be handled by *only one* receiver, but can be picked up by any of the connected endpoints. This type of connectivity is frequently seen by dispatcher-type services, such as server front-end services, which dispatch jobs to a pool of worker processes which actually handle the job. In this configuration, the precise worker executing the job is unimportant, merely that the job is handled as soon as it is possible to do so.

As the data in the buffer only needs to go to a single receiver (any of them) there only ever needs to be a single instance of the buffered message; any more than this would be pointless, as once a receiver has picked up a message, there is no need for the remaining receivers to have access to the data. A naive solution for this might include buffers at the receiver end, rather than the transmitter side, but doing so would preclude demand-based retrieval, thus pushing data onto what may already be a loaded receiving process, causing needless wait times.

Even in the fairly simple example shown in Figure 3.6, there are additional implementation specific complications.

The networks that have been discussed so far have been *open*; open networks have no in-degree greater than 1, and thus, can be described fully by tree structures. In Figure 3.10, the network is now *closed* at node 4, and in doing so brings a number of interesting problems.

If messages are immutable and transferable, such that a given message at 1 is eventually passed to 4 via the paths $p' = \{1, 2, 3, 4\}$ and $p'' = \{1, 2, 5, 6, 4\}$ not only will there be two apparently identical messages received at 4, but also there will be an out-of-order message

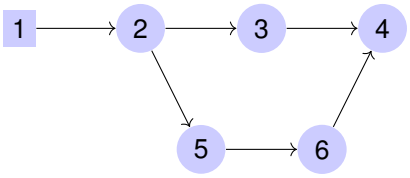


Figure 3.10: *Branching* and *Merging* communication paths in a process network. Note that the semantics of handling the plural in-dimension of Node 4 can become complex on their own, without the complexity of the rest of the system.

received at 4, caused by the differing path lengths ($length(p') = 4$, and $length(p'') = 5$).

Therefore, if the process performed at node 4 is sensitive to message order, a considerable number of messages may be required to be queued up for processing from path p' while waiting for p'' to catch up. Unfortunately, the larger the total data payload is, the more extreme this buffering requirement becomes, so while this situation is surely unavoidable in some cases, if the network is to be generally buffer-light, individual messages should be as self-contained as possible.

In summary, beyond the rules for handling messages in the network itself, there is also a need to define a way to handle the *inbound* and *outbound* messages at each node in the graph, as this dictates many of the characteristics of the network beyond each node.

3.3.4 Lock-Free Data Structures

So far, this text has focused on the network and messages, but some discussion of the data structures which could be used to handle the message passing and storage required to realise these designs.

Lock free data structures offer an alternative to traditional lock-based multi-process-access or multi-thread-access structures. Instead of guarding sections by semaphore or mutex operators, they instead use a number of processor-specific atomic operations to ensure that the structure is preserved in a valid state at all times.

This has advantages both in speed - atomic operations are often single-cycle, rather than mutex or semaphores which often run for multiple cycles - and concurrency - as the access is mediated by operations which often have no upper bounds on the number of concurrent actors, and therefore are desirable for the kind of high-speed data switching operations thus far discussed.

Traditional multi-access locking structures generally follow the access methods shown in Algorithm 1 and 2, which use mutex-type structures to lock sections off from multiple threads while ‘dangerous’ operations are performed.

Variations on these use semaphores (also known sometimes as counting mutexes) to restrict

Function `AccessSharedStructure()`
Get reference to shared structure;
Acquire Lock
Get a reference to mutable data;
Mutate any data required in the mutable data block;
Store mutated data;
Restore valid state;
Release Lock
Release any temporary resources;

Algorithm 1: A single atomic section encompassing the entire function. This is pretty much the worst design for implementing this kind of access as it precludes any interleaving of operations for the entire runtime of the function.

access to a small number of actors rather than just a single one.

Function `AccessSharedStructure()`
Get reference to shared structure;
Acquire Lock
Get a reference to mutable data;
Restore valid state;
Release Lock
Mutate any data required in the mutable data block;
Acquire Lock
Store mutated data;
Restore valid state;
Release Lock
Release any temporary resources;

Algorithm 2: Two atomic sections provide an opportunity for other accesses to the shared structure while this one is mutating the data in the block it requested. This is better, but still requires that the entire structure be 'owned' by this process to proceed.

By contrast, lock-free structures attempt to ensure that the only operations performed on a structure that are considered 'dangerous' are those that can be done entirely atomically in a single instruction on the processor.

There are only a handful of truly atomic operations that can be done; gcc (as of 4.4.5) supports three types of access via memory barrier mechanisms; *fetch-then-muate*, *mutate-then-fetch* and finally *evaluate-then-execute*.

The first two types have the form (in this case, for an *add* operation) `__sync_fetch_and_add` `__sync_add_and_fetch`, and are supported for the `add, sub, or, and, xor` and `nand` functions. Together, these form semantics which align with pre- and post- increment or decrement operations familiar to C-like languages.

By way of example, a guaranteed pre-atomic-add (aka. pre-increment) would involve a call to `__sync_add_and_fetch()`.

While these functions are clearly useful for lock-free operations, (as they require no additional boilerplate to ensure atomicity) the real enablers are the *evaluate-then-operate* type

functions.

- `__sync_bool_compare_and_swap`
- `__sync_val_compare_and_swap`
- `__sync_lock_test_and_set`

These functions allow decisions based on the state of memory to be made entirely atomically while also modifying the state of that memory. The classical example for where this can be used is the `compare_and_swap` operation, which is often used to implement mutexes themselves by restricting the acquisition or modification of a variable in memory (a guard variable) without the variable being in a declared state.

Function `AccessSharedStructure()`
Get reference to shared structure;
Atomic Get mutable block and set to new valid state;
Mutate any data required in the mutable data block;
Atomic Set shared structure to new valid state;

Algorithm 3: A simplified overview of lock-free access to shared structures. Note that the lines marked **Atomic** are performed as a single (and thus un-preemptable) operation, and are processor specific.

Consequently, through these mechanisms, a lock-free structure can be formed that performs as described in Algorithm 3, allowing *potentially wait-free* access.

While a *lock-free* structure are entirely possible using these operations, the behaviour of such implementations will be in one of two classes, either truly *wait-free* operation, or merely in a *lock-free* manner.

Wait-free functions allow a calling process to continue irrespective of the internal state of the underlying data structure, whereas *lock-free* functions may cause non-deterministic delays when called, but attempt to minimise the delay they cause (generally) by keeping the caller resident throughout the process.

Depending on the particular requirements of the process, either access may be preferable, but only the application can know which is objectively *better* in most cases.

For the purposes of buffering data between processes asynchronously, the wait-free approach is preferred, but not required, as traditional transmitting processes already expect that their output channels may halt transmission until the receiver is ready to handle the data.

If wait-free structures are available for the buffers, however, they can be used to reduce the output jitter as seen by the process to zero, excepting cases where throughput limiting is required.

3.3.5 Summary

Naturally, with the constraints of only using atomic operations to enforce structural validity, the implementation of a lock-free structure can easily become very complex. Thankfully, the libLFDS[33] library provides a number of pre-made lock-free structures ready to use, including a ring buffer and various queue implementations - both perfect for use in buffering operations.

However, for the purposes of the prototype, the dominant overheads are more likely to be the actual message transfer mechanism, as the number of nodes in test setups will tend to be quite small, so address table and forward list lookups should be close to as fast as they can theoretically be in most cases.

3.4 Map and Reduce

When faced with a large data set, it is not uncommon for approaches to process this data to look to multicomputing frameworks such as Apache Hadoop[58] to distribute the processing load to a cluster of machines configured to execute jobs from one or more dispatchers.

For Apache Hadoop (henceforth referred to simply as 'Hadoop') the key processing primitives are the *map* and *reduce* operations, from which all operations are derived in this class of computing.

The basic function of the *map* function is demonstrated graphically in Figure 3.11 along with its pair function reduce in Figure 3.12 - it is worth noting that the exact order of operations applied does not matter for the purposes of *map* or *reduce*, as normally the input data is normally immutable, requiring the function or framework to build an entirely new data set for the return. Sometimes this new data set in-place replaces the original data set reference or storage, but during the operation there are two distinct sets of data in play, the input and output.

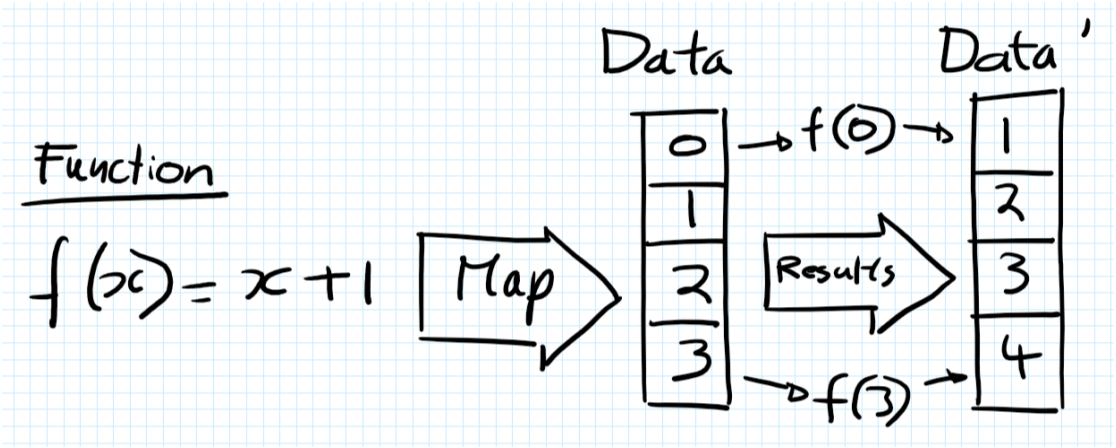


Figure 3.11: A graphical representation of the 'Map' operation.

Taking a function input, 'Map' applies this to each element of a data set returning the results

of this operation with the indices unchanged. This function needs not only relate to a single element, but may require multiple elements in the input set to produce the output set - this means that implementations commonly use immutable data structures for the input and output sets, such that data cannot be mutated by one function which would immediately affect another, as this would cause the *order of processing* to matter for the results to be consistent between runs.

This can become a problem for extremely large input sets, if only from a memory allocation perspective; the requirement for the output set storage to be separate from the input set means that naive implementations can end up requiring double the storage space to operate correctly. Some implementations may reuse unchanged data from the input set, only storing the changes in the new output before merging them back into the input set for an in-place operation, but doing so requires that the function being applied only partially mutates the data.

Without the average case for these operations only touching a subset of the input set, the change tracking requirements of the output set end up causing its overall storage requirements to be much larger, which will result in unpredictable performance due to differing complexity of writing a change as to not, and having two independent look up and storage systems in use.

Taking a supplied function, a 'Reduce' applied this to the entire data set of $size = n$, (or sub-collections therein) and returns the result as a collection of $size <= n$. In many cases, this operation will result in a smaller output data set than it was supplied, hence the name 'Reduce', but this is not always the case, as some 'Reduce' operations will inherently be unable to produce a smaller set.

While the functions performed by *map* are both injective and surjective (thus form a bijection) *reduce* performs non-injective, but surjective functions; by taking the input set and applying a function that conforms thus it produces output sets that either match, or ideally, are smaller in size than the input - hence the term *reduce*.

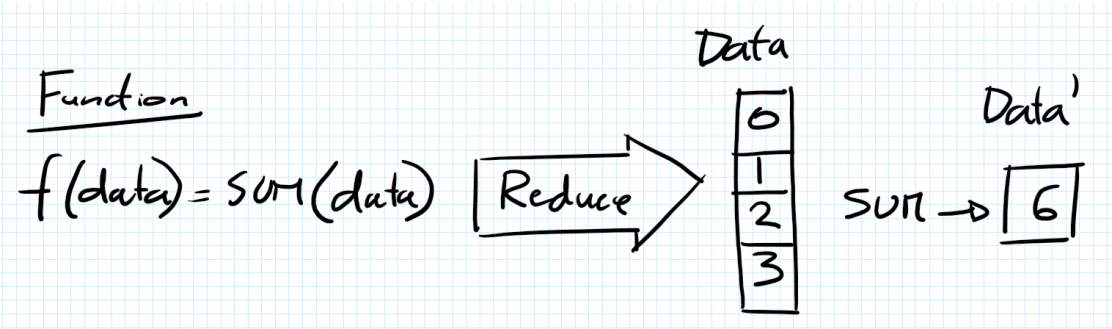


Figure 3.12: A graphical representation of the 'Reduce' operation.

In a computational linguistic sense - or indeed, any discipline with extremely large data

sets - the requirement for these two functions to have an immutable input set becomes very problematic. With the limiting factor of available RAM in any given host, it quickly becomes impractical, if not impossible to perform any operations on the input data without having to resort to dividing, simplifying, or otherwise altering the data such that it will fit alongside the output copy, which can be up to and including the same size as the input set.

Oftentimes, this is achieved by first encoding the input set in to a representative form - the exact nature of which depends on quite what operation is being performed - then translating the output back in to a form useful to further processing. With a multi-stage process (a not uncommon state) this can mean that multiple translations are required on both the entry and exit to and from a process; this can be visualised as shown in Figure 3.13.

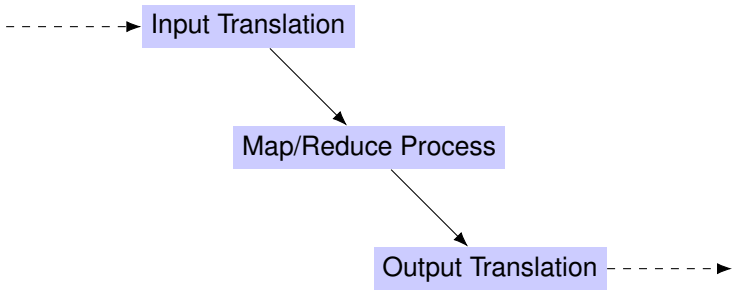


Figure 3.13: Sending data through a map/reduce (or otherwise highly parallel operation) that requires that the input set be immutable often requires additional translation stages to ensure that the data can fit in working memory.

Thus; if the data in question is highly co-dependant, the total amount of parallel processing is limited by that dependence. If we assume we have a data set with a given number of elements our ideal parallelism - assuming a machine capable of infinite parallel processes - is also that same number.

$$\text{parallelism} = \text{elements} \tag{3.1}$$

However, if each element requires some number of other elements to have the function executed on it, the total parallelism becomes:

$$\text{parallelism} = \frac{\text{elements}}{\text{dependants} + 1} \tag{3.2}$$

In practice, this means that the minimum number of sequential passes over a given data set to get it to a new valid state becomes the inverse of this.

$$\text{sequential\_passes} = \frac{\text{dependants} + 1}{\text{elements}} \tag{3.3}$$

A clear case where this parallelism model is apparent is where a GPU is used. Processing data on a GPU is not as straightforward as running the same algorithm on a multi-core general purpose processor; both the communication and memory architectures of GPUs have their own particular quirks that need to be accounted for.

The next few sections will detail a case study using a high-performance GPU for processing concordance lines from a large input dataset - one that almost entirely fills the GPU's on-board memory - and explore where and how GPU computing techniques achieve the performance they are known for.

3.5 GPU Case Study - GPU Sort

Corpus data is used in many areas of Digital Humanities, Natural Language Processing, Human Language Technologies, Historical Text Mining and Corpus Linguistics; increasingly, however, the size of corpus data is becoming unmanageable through traditional means. Taking digital humanities as an example, national and international digitisation initiatives are bringing huge quantities of archive material in image and full text form directly and quickly to the historian's desktop. Processing such data at speed, on the other hand, will almost certainly exceed the limits of traditional database models and desktop software packages.

Similarly, the "Web as a Corpus" paradigm has brought vast quantities of Internet-based data to corpus linguists. However, any search or sort of results from these rich datasets is likely to take from minutes to hours to days using desktop corpus tools such as WordSmith Tools[60] and AntConc[1] as the data in question continues to grow.

To address the problems of handling massive data sets, international infrastructure projects, such as CLARIN[26] and DARIAH[27], are emerging with support for these large corpora under the umbrella of 'big data'. In corpus linguistics, researchers now have access to tools such as Sketch Engine[48] and the family of BYU Corpora[9], which aim to support pre-compiled billion-word corpora. In both cases, these systems are remotely hosted and are also not particularly easy to configure for specialist datasets, as is more and more the case in the fast moving corpus linguistics arena.

Recently developments have produced semi-cloud based systems; GATE[21], Wmatrix[59], and CQPweb[45] all provide interfaces which can provide users with local access to large data sources. However, the installation and configuration of such systems is far from simple, making them inaccessible to most social science and humanities based scholars.

Hence, there is still a need to investigate processing efficiency improvements for locally controlled and installed corpus retrieval software tools and databases - the researchers' desktop continues to be the primary source of computing resource used. Core tasks such as corpus

indexing, calculating n-grams, creating collocations, and sorting results on billion-word databases cannot feasibly be carried out on desktop computers within a reasonable timescale using traditional techniques - even those which utilize the common multi-core processors in those desktops.

Rather than leverage the compute power of the continually growing distributed computing frameworks, we approach these problems by targetting the currently underused resources of the local machine. GPUs are common, and capable coprocessing devices built in to almost every device - even embedded systems now contain discrete GPUs - and these are being used to great effect in other fields of science, but they often do require a change in how data is handled due to their very nature.

In recent years, high-performance, general-purpose graphics processing units have become increasingly available to the scientific community, and projects utilising them have been met with considerable success as described in Deng [13] and Melchor et al. [37] and Sun, Ricci and Curry [49]. Unfortunately, their use in corpus linguistics and natural language processing has been limited at best, and many areas of their uses have yet to be explored.

With corpora exceeding the multi-billion-word mark, even these measures are unable to complete experiments within reasonable time, often spanning days of operation [57]. In addition, enhancements designed for other areas of computing, e.g., Cederman and Tsigas [10] and Rashid, Hassanein and Hammad [44] have proved to be not well suited to corpus processing.

3.5.1 Issues with GPU Hardware

GPUs are extremely good at running vast numbers (in the thousands) of parallel processes on independent data. The problem comes when processing that data requires other elements in the same data set; the architecture currently used in these devices means that the view any given processor has on memory is unlikely to be the current, complete, coherent one, but a shadow of some previous state.

Exactly how this memory is arranged is shown in Figure 3.14, with multiple layers of caches, the precise view a single processing element has on the overall state of the GPU memory is very questionable at any given moment. This is compounded by multiple processing elements using the same cache as the layers get closer to the main GPU RAM.

To ensure memory coherency across large input sets, there need to be certain ‘checkpoints’ (of sorts) which allow the CUDA framework to re-synchronise and consolidate the memory at the various cache levels. Note that the memory *reads* use the highest level without any local changes, but as soon as a *write* is performed, the first-hit cache records the new value, so locally, changes are correct, but crossing a cache boundary will give incorrect results, quickly leading to semi- or non-deterministic behaviour.

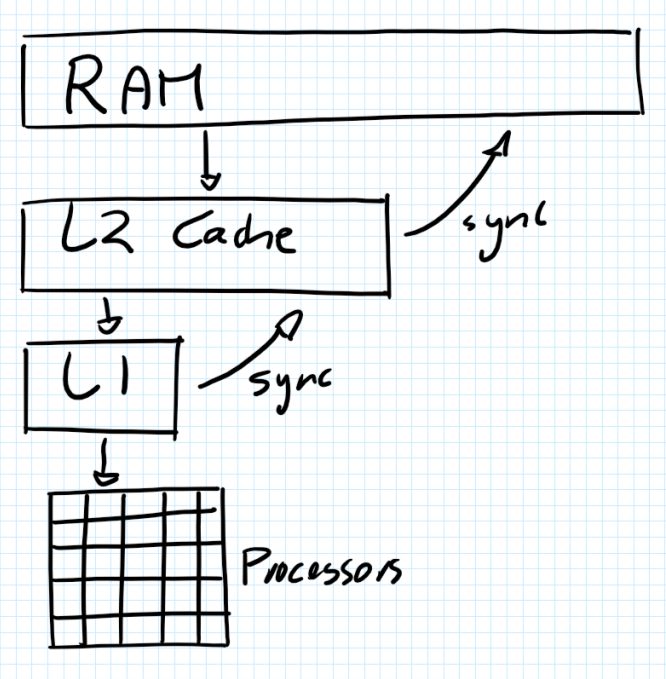


Figure 3.14: The layers of caching in a GPU; Note that *sync* operations are likely not to occur very often during normal operation, unless the software mandates it, and to ensure coherency, the processing on the synchronising elements must briefly halt. This structure is replicated for each group of processors in the GPU.

3.5.2 Expermiental Methodology

... began to diminish and soon	there	were no more visitors Madame ...
... as though it had been	there	for months He even went ...
... of declaring that as yet	there	were no signs of decomposition ...
... the stairs were distinctly heard	There	was silence for a few ...
... ready to go downstairs when	there	appeared before her her son ...
... the terms of this agreement	There	are a few things that ...
... agreement See paragraph C below	There	are a lot of things ...

Figure 3.15: A sample of the input set used for testing the sorting algorithms, truncated to fit this format. The actual prefix and postfix strings were at least 10 words long each

To evaluate the potential gains of using General Purpose Graphics Processing unit (GPGPU) techniques for corpus retrieval operations, it was decided the focus should be on one of the most common and time consuming tasks that corpus linguists need to perform; namely that of sorting the concordance lines generated from the output of another process, such as those from a database query.

Once concordance lines are extracted or displayed in a corpus retrieval system, analysts need to be able to identify language patterns in the results set. Due to the large number of results, the concordance lines cannot reasonably be read manually, and so some pre-processing is required typically involving a sorting stage.

Most concordancing tools, such as WordSmith Tools and AntConc, can perform a multi-level sort of the results based on the preceding and/or following words. However this can be a very lengthy operation, especially when many hundreds of thousands of concordance lines emerging from large corpora require processing; this can be especially bad if the output set is large enough to span multiple memory boundaries, and unfortunately this is often the case.

To generate a dataset to process, the published BNC frequency lists of "Word Frequencies in Written and Spoken English" (Rayson et. al)[28] were used, which in turn were used with a corpus generated from the Gutenburg Project books data[29] to generate CSV input sets as shown in Figure 3.15.

These were loaded into memory in full, and stored such that the entire concordance was kept in RAM. While this may seem sub-optimal, this was done in an attempt to present data that represented the performance of the GPGPU device, rather than memory usage tricks. Additionally, the batch-processing style of operation used in GPGPU computing limits our ability to do most traditional sorting techniques for large data sets, such as an external merge sort, as the GPU available does not support the recursion depth required to process this data[30].

Following the above procedure, it was possible to reduce the problem to an entirely data oriented issue and avoid the characteristics of disks, buses, networks and other hardware components interfering with the performance measures.

```
Let collection = Unordered array of values to sort ;
Let actions = 0 ;
repeat
    actions = 0 ;
    for index=0 .. n, step 2 do
        if collection[index] >collection[index+1] then
            Swap collection[index] and collection[index+1] ;
            actions ++ ;
        end
    end
    for index=1 .. n, step 2 do
        if collection[index] >collection[index+1] then
            Swap collection[index] and collection[index+1] ;
            actions ++ ;
        end
    end
until actions == 0;
```

Algorithm 4: The 'Swap Sort' algorithm, whereby even, then odd pairs of elements are tested to determine if they should be swapped, and if so are. Each swap is counted, and if no more swaps are found, the algorithm breaks out of the testing loop. This is normally a terrible way to sort a dataset, but with the abundant processing in the GPU, the technique proves quite powerful.

A preliminary investigation of different sorting algorithms concluded that most are very data-copy sensitive (requiring many short batch operations and many host-device memory copies),

so instead of using particularly elegant algorithmic techniques, the simplest sorting technique with the maximum parallelism was selected: the swapping sort (See Algorithm 4).

In the swapping sort, concordance lines are directly loaded into memory on the graphics card and processed in place by comparing each line to its immediate neighbours in the input set, and swapping the entries if they are incorrectly ordered; this is repeated until the entire set is sorted. While this technique would be extremely inefficient on a CPU (by modern sorting algorithm standards), it works impressively well on a GPU, as we can perform large batches (over 27,000 entries, on a nVidia GTX Titan) at once.

The specifications of the machine used for these tests is described in Table 3.1, but the exact specification for the hardware is not particularly critical for achieving similar results. The use of a GTX Titan or better is recommended should this experiment be repeated, as older cards have smaller video memory areas, resulting in higher instances of copying to and from the mass storage device (hard drive or equivalent). The SSD in particular is not required, but was used for these tests to expedite the test duration through eliminating the delays normally incurred through using mechanical disks; even with the sporting speedup of the GPU, the bottleneck in this configuration would always be the load and unload phases due to the limited copy bandwidth of the GPU[31].

CPU:	Intel “Sandy Bridge” i7, desktop edition (quad core with hyperthreading support).
GPU:	nVidia “GTX Titan” graphics card, 6GB video memory.
RAM:	6GB Triple Channel memory.
Disk:	A generic 64GB 6GB/s SSD.

Table 3.1: The specification of the machine used to perform the tests.

3.5.3 Results

Running the swap-sort algorithm on the GPU with the provided data yielded the results shown in Figures 3.16 and 3.17[32]. As can be seen in these two figures, the GPU accelerated sort consistently beats the sort on a normal CPU excepting in cases where the input set is extremely small.

Below an input set size of 2000 concordance lines, the CPU has a slight advantage, as the GPU has a small delay involved with deploying CUDA kernels[33], presenting an absolute minimum throughput. However, above 2000 concordance lines the GPU is several orders of magnitude faster than the CPU, and remains consistently better throughout all the other test cases.

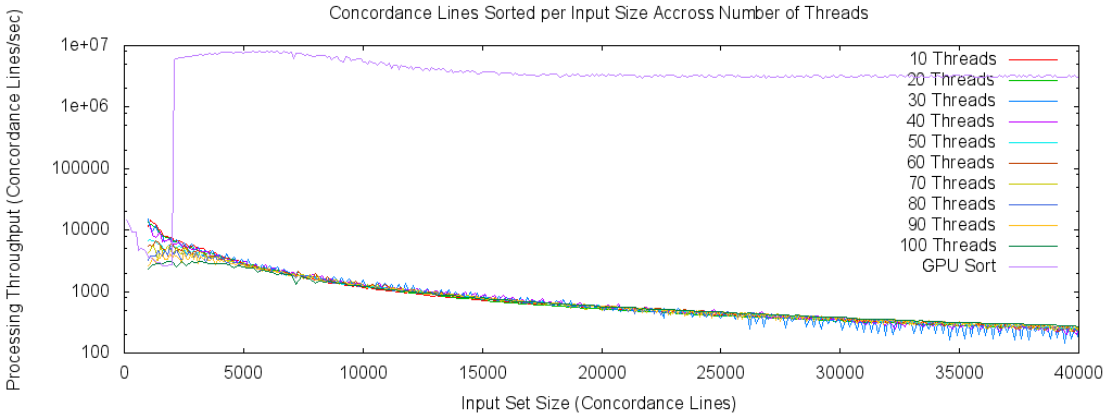


Figure 3.16: The measured CPU and GPU performance measurements shown on the same axis. Beyond 40,000 concordance lines, the sorting technique took so long to complete on the CPU as to be useless, while the GPU continued to perform exceptionally well. Note that the y-scale is logarithmic.

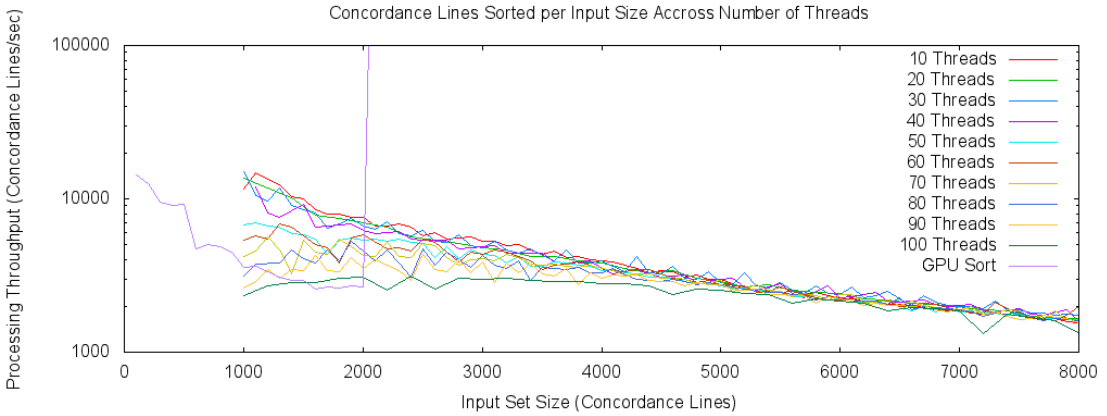


Figure 3.17: The first portion of Figure 3.16, showing the initial CPU advantage for very small numbers of concordance lines.

3.5.4 Discussion and Conclusions

This experimentation clearly shows the power of the parallelism inherent in GPU hardware, along with the ability of the map/reduce paradigm to maximise the potential performance in this hardware. Whereas CPU-only based approaches quickly become cumbersome above 40,000 concordance lines, the GPU approach - even using such an inefficient sorting algorithm - carries on performing far better up to an beyond 10 million lines.

The memory coherence issue is problematic for traditional sorting methods, with GPU memory intentionally having inconsistencies introduced to increase the processing speed of individual processing elements, there needs to be care taken to ensure the validity of the processed data. At the time that this experiment was carried out, CUDA had yet to provide a mechanism for long-running processes on the GPU to communicate directly with the host runtime, and this prevented

any further experimentation with techniques involving external long-term-storage, however with a more up-to-date GPU and CUDA runtime, it should now be possible to achieve this kind of live-processing, provided that any scheduling and/or locking issues can be overcome.

The processing used for these tests is best described as 'out-of-data-path' processing, as the data would not normally be processed where it has been for this experiment; Normally one would expect that moving the data further from where it is stored would result in slower overall performance, but with the immense processing power available in a GPU, once in place the gains more than make up for the longer overall data path, as was also the case for Deng [13] and Melchor et al. [37] and Sun, Ricci and Curry [49].

This style of processing - taking an operation and vastly increasing the number of parallel instances of this operation to get increased performance - has been shown in this case study to have significant gains, even when (or arguably *especially when*) the actual operations in question are extremely simple.

This has been known in the literature for some considerable time, but only in recent years have the processors with enough actual parallel capability been available; although even in situations where a multiprocessing unit is used, if the chains of operations are reasonably distinct, some of the gains here can also be realised.

3.6 CPU Bugs - Meltdown and Spectre

During the course of this thesis - specifically in the early part of 2018 - it became apparent that there was a serious problem with the behaviour of certain processors (generally Intel, but also AMD and ARM), whereby attackers were using the characteristics of the operating system memory access to determine the *value* of the memory therein.

These attacks were known as Spectre[30] and Meltdown[34], both of which were significant enough to warrant public visibility and their own informational site[34].

In the case of this work, however, they are important as the patches that came to fix these issues altered the way in which memory could be accessed and written across the board[35]

The specific paths the operating system took after the patches removed the differences in execution time between accessing valid and invalid memory areas - this would normally not be a problem, as most processes naturally use only valid memory areas, but in cases where memory access restrictions are put in place using hardware support, this can result in noticeable slowdown.

As has been discussed in this chapter (Section 3.3.2), there are techniques which endeavour to use the memory management hardware (often referred to as the MMU or *Memory Management Unit*) as a gatekeeper for message access, and it should be noted that these attacks and

subsequent modification to the system behaviour will have a negative effect overall, and prior attempts to characterise the systems have been invalidated by these changes.

3.7 Summary

To make meaningful inroads into processing large amounts of textual data, it is no longer simply enough to throw standard general-purpose processors at the problem. Simple parallelism is no longer enough to deal with the complex input data without risking the coherency of the output, and while locking or transactional data structures (such as database systems) provide one way of handling this in a practical sense, they greatly limit both the flexibility of the solutions presented, and the overall performance.

Having a single point of coordination is no longer a viable solution to this problem, and instead there must be a distributed method of dealing with the *flow* of data through analysis tools.

In the next chapter, the design of a solution using the techniques explored here is explored and constructed.

Footnotes

[22]At the time of writing, details of the xMos XCORE modules can be found at https://www.xmos.com/products/#general although the website seems to be changing.

[23]`signal` manual page, http://man7.org/linux/man-pages/man7/signal.7.html

[24]According to the pipe man page; https://linux.die.net/man/7/pipe, the pipe buffer is 65536 bytes since Linux 2.6.11 - ie. *unsigned short* long in C

[25]System on Chip

[26]See https://www.clarin.eu/ for the current project status

[27]See https://www.dariah.eu/ for the current project status

[28]http://ucrel.lancs.ac.uk/bncfreq/flists.html

[29]http://www.gutenberg.org/wiki/Gutenberg:The_CD_and_DVD_Project

[30]When this experiment was run, the CUDA framework did not support the required features for a merge sort, namely the recursion depth and the dynamic instantiation of new kernels. This is now possible, and further research inroads have been made by other authors into other sorting techniques.

[31]This particular experiment was carried out in 2014, since then the available copy engines in nVidia GPUs has increased significantly - a modern reproduction of this methodology would have far less load and unload delay, and may also be able to stream the data in as required from a larger backing store. Unfortunately the hardware available at the time was incapable of this, despite efforts to the case.

[32]The data for these plots as well as additional data can be found at http://johnvidler.co.uk/academia/cmlc-2014/

[33]A CUDA kernel is the GPGPU equivalent of a CPU thread

[34]https://spectreattack.com/

[35]The patches themselves included both operating system modifications *and* microcode patches to the processors themselves

Chapter 4

Design

Fundamentally, what seems to be missing from the available interactions, both on normal general purpose hardware as well as specialist hardware (such as GPUs, FPGAs, and others) is the ability to describe communication paths in anything other than a one-to-one relationship.

This limitation presents problems when, for example, the application in question wants to send data to multiple receivers. While this is more than possible using pipes and middlewares, there always remains the problem of mediating those connections manually at the application level.

Rather than encoding the semantics of the interactions in the processes themselves, an arguably better alternative approach would be to encode the interactions in the system itself, releasing the processes to be simple data-flow processors, rather than having to also deal with the connections. What follows here is the an attempt to describe the idealised interactions between processes, and how this would function with no restrictions imposed from hardware or software constraints. As such, this should be viewed as the *design target* for the subsequent chapters, but moving beyond this design real-world problems will have to be taken into consideration.

4.1 Outline

As we have so far seen, at the most basic level, current process interactions trend towards a 1-in, 1-out model - UNIX pipes being the classic example for this - and while this is extremely powerful, allowing processes to be connected together, it does limit what can be done when the interaction requirements fall outside this 1 in-and-out degree model.

Therefore, let us presume a mechanism whereby processes can interact via n-in and n-out paths; in this case, the possibilities expand to encompass far more interactions. Multiple passes through a single instance of a given process cease to be a batch operation requiring that the pipeline be re-instantiated for each pass, and become a looped link from one process to itself; multiple input sources to the same process *graphs* become a simple additional input process

and new link, rather than having the entire tooling repeated.

Ultimately, the goal is to avoid the need to represent non-linear communication paths as sequential linear process chains.

4.2 Design Concepts

To provide a mechanism through which messages between processes can be marshalled with similar techniques as used for networks at scale, it makes sense to separate the routing of the messages from the nodes sending the messages in the first place. Without this separation, it is difficult to control the network as is required.

From a design purism point of view; each process is modelled as an n-degree node, each arc connecting to each other process by a unidirectional link. The machinations of how this is actually achieved may differ from this - take for example the case where links are on different hosts, in this case there must be at least one network link between the two devices, but to the application nodes this is not seen as any different to a local link, in much the same way that sending a network packet from one host to another may actually go by any number of paths, but to the processes at either end, the transfer is seamless.

This is a divergence from normal POSIX[36] or System-V[37] interface whereby any connection between two processes (virtual or otherwise) is modelled as a bidirectional link. This is significant for two reasons, one; the semantics of the upstream link need not be the same as the downstream link, and two; the process may never require there to be a bidirectional link at all.

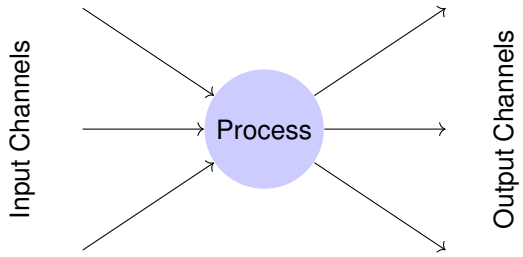


Figure 4.1: A simple view on a given process from a connectome, note that links are directional.

4.2.1 Asymmetric Network Links

An interesting characteristic of symmetric network links in IPC systems is that they, by topology, negate certain connectivity options. Taking a bidirectional link and attempting to handle multiple receivers quickly leads to routing complications requiring various forms of message tracking and routing overheads. Rather than deal with this, this design uses simple unidirectional links, and in the case where bidirectional communications are required, two such unidirectional links can

be constructed in the network graph.

4.2.2 Unidirectional Network Links

The original inter-process links were modelled around the same concepts as serial links in early computers; byte-per-byte transfer and bidirectional, but this is no longer required to be the case. Rather than have a full bidirectional communication link between nodes in the network, having a single unidirectional link paired with a minimal flow control back-channel provides a far simpler, but in practice, no less capable link. Indeed in many applications, the transfer bias for any given link will be tilted one direction or another. with a serving-node providing overwhelmingly larger amounts of data than the receiving peer, for example.

This particular design is somewhat borrowed from schemes such as MPLS[38], whereby network routes are specified before they are used, eliminating the need for the routing layer to calculate the next hop for each packet, relying on a precomputed destination list based on the source of the packet at each hop in the stream. In both the MPLS case and here, the packets still do contain the destination address allowing routers to perform destination routing, but in reality, knowing just the *source* address is enough to perform a look-up for the entire route at any step; if the source is 'X' then forward to port 'Y'.

As this design follows macro-network design themes fairly closely, the mapping between micro- and macro-operations offers the possibility of using LAN (or larger) hardware to handle the routing roles currently embodied in the software routing component.

For example, *Multiplex* and *Map* operations can be achieved at the packet level via multicast groups; clients joining a multicast group receive broadcast packets by default, but many routers do support anycast on a multicast group, picking a target from any of the connected clients. Furthermore, the broadcast messages (to a given group) provide the same distribution as the *Map* operation, forwarding a copy of the message to every connected client - and as both of these operations are supported by hardware acceleration in the routers themselves (in many cases, although indeed, software routers are still numerous) the throughput over network links should still remain high, despite the operations being encoded within.

It should be noted that while multicast groups offer the best way of describing the required behaviour for *Map* operations, the original IPv4 'broadcast' mode format would also achieve this, although through a much broader brush. Every connected client would receive the forwarded message, with no filtering at the local level, restricting the switch or router to representing a single *Map* operation, rather than being able to support multiple parallel, but independent *Map* operations.

All of the LAN networking components, however, are secondary to the internal inter-process 'wiring' that this design primarily addresses, but the design similarities are referenced here for

completeness.

4.3 Architecture

In the next few sections, some discussion of the architectural design and decisions of individual components of the system are discussed.

4.3.1 Stream-based approaches to Data flows

Rather than taking a data-oriented approach to dealing with large input data sets, the GraphIPC design takes a stream-oriented approach instead.

In large scale networking, it is not uncommon to use schemes which take all packets with a particular field value and route them based on a pre-loaded policy, such as is the case for MPLS.

This design pattern can be leveraged to route messages in the graph network without the transmitter having to know the destination in advance, simply blindly transmitting to the router and leaving it to handle the actual foward lookup and path. This greatly simplifies the packet and node software design.

The other end of this means that the *receiving* node will have a source address but no destination to key its actions from, resulting in nodes which have behaviour effectively defined by their route rather than their address, somewhat blurring the lines between the router and the end node.

4.3.2 Router

At the core of this design is the router - as might be expected, this component actually performs the forwarding operations needed to actually get packets from source to destination, but further to this basic requirement, the router (or routers, in a multi-router configuration) are the only components in the graph which have the network connectome. While clients aware that the network is present may also record their own paths for internal bookkeeping, the router(s) are the only component required to hold the network edges (functionally, the actual network connections).

4.3.3 Nodes

Rather than referring to a the vertices of a communication network as *processes* - as they may often be - instead they should be referred to as *nodes*, as they can also be virtual connection points existing only in the configuration of some other component. This can most often be seen in the *router* - cases where streams are joined together to be sent to a single endpoint are

technically vertices unto themselves, but cannot be described as *processes*, despite having some reception and forwarding capability.

4.4 Interaction with other Linux Processes

Providing a new method for interprocess communication is only as useful as its interaction with existing tools, otherwise the entire system would have to be re-engineered to use the new communication layer.

4.4.1 Tool Interoperability

Beyond the trivial interactions with corpus data, the methods for getting data in or out of a storage system become increasingly complex. Data coherency is one area where these issues are particularly prevalent, but another equally important area is that where the stages in a data transform sequence meet.

Large frameworks work around this issue by mandating a single data transfer format, usually with meta-data that includes the specifics. However, moving beyond the bounds of a given framework often requires handling the format changes at the edges of the tools in question, and that quickly becomes difficult (at best) to manage effectively. There are even cases where the tooling agrees on an particular data format, but the encoding *within* that format differs, resulting in wildly inaccurate results.

As part of an exploration of this problem space, a partial solution to the encoding and formatting problem became evident. By embedding additional structured data inside existing data formats, it become possible to include the otherwise missing properties required to handle the formatting transforms to allow tools to interact.

In the experiment, this was achieved through the use of a tool-wrapping-tool which attempted to validate that the data that was being passed to the enclosed process was in a format that the tool could accept, or at least, to explicitly reject any attempts to input invalid formats.

Taking this concept further, it became evident that it should be possible to perform some of the transforms at the edges of the wrapped processes automatically, provided that the data could be streamed in such a way that performing the transform would make sense at all. Naturally, if the particular format requires that *all* data be transferred before processing can begin, the memory overheads in doing this are going to be quite high - consider a graph-type node/edge storage medium, requiring all edges before the graph can be built - but simpler incremental formats should be possible to do in-stream.

4.4.2 Structured vs. Unstructured Data

When importing or exporting data, a process can use any number of different formats, but irrespective of which particular one is chosen, they will fall in to one of two categories; either *structured* data, or *unstructured* data.

Unstructured data is both the easiest to understand, and (possibly therefore) the most common form seen. Comma Separated Value (CSV) records are common, using a predefined symbol as a field delimiter, and relying on the receiving end agreeing on the field offsets to ensure that the correct data is transferred.

```
field1,field two,"quoted string field"
test,line1,line4
```

Figure 4.2: An example of a CSV file - note that the field definitions are unknown for any given 'column', and would require further data to correctly parse.

However, because of the inflexibility and ambiguity of formats such as CSV, there is an increasingly popular trend to encapsulate all transferred data in structured formats such as XML[55], YAML[18], JSON[16] and others. These structured formats have then themselves spawned other efforts to wrap the structure in to binary formats - BSON[24] and MessagePack[20], for example - to save on bandwidth and parsing between programs, rather than the original human-to-machine interaction that they were intended for.

JSON:

{"field1":true,"field2":0}

MessagePack:

```
82 a6 66 69 65 6c 64 31 c3 a6 66 69 65 6c 64 32 00
```

Figure 4.3: A simple message encoded in both JSON and MessagePack formats. Note that the MessagePack data is hex-encoded binary, with whitespace added for clarity.

4.5 Interactions

As the design presented here is a superset of the traditional UNIX pipes system, processes designed to work in such an environment can be fooled into using the graph network the same way, invisibly (to the process) receiving data along the more complex paths. To support these processes, a number of discrete operations are required, as the functions they provide are not possible with the 'bare' UNIX processes; these include the now common *Map* and *Reduce* primitives, but also describe *Mux*, *Demux* and *Bus* operations. These together form all the basic requirements for communication in the graph without requiring that the process does anything other than handle regular UNIX pipes.

The differences between some of the individual operations can appear subtle, as they do extremely related things, but do so on *different aspects* of their inputs. *Map* and *Reduce* do not preserve the addresses of the input packets, instead performing their respective operations on the data as though the output is a single input or output stream, as seen by connected operations. By contrast, the *Bus*, *Mux* and *DeMux* operations preserve the any addressing, but determine the internal routing requirements of the flows - performing each particular mapping operations without altering the data. The precise function and reason for this distinction will become clear as each is discussed.

In some exceptionally rare cases it may be useful to perform one or more of these operations without there being a particular processing function - load balancing may be once such case. In these instances, the operation is done *in the network*, effectively being applied by the router; although technically it could be applied as a form of pseudo-process, whereby it exists as a node in the graph, but does not have a processing function; a blank process, if you will.

However, doing this in the network itself in cases where software routers are being used - including the host-internal router, even in cases where it exists as a kernel module - should be discouraged, as it will place processing loads on the router, potentially lowering the overall throughput of the graph as a whole as the router deals with the received data and logic. If, alternatively, this is implemented as a hardware device - either host-locally as a specific piece of switching hardware, or externally as a discrete router with special hardware - the throughput reductions may be undetectable due to the hardware support, so could be used in these cases.

4.5.1 Bus

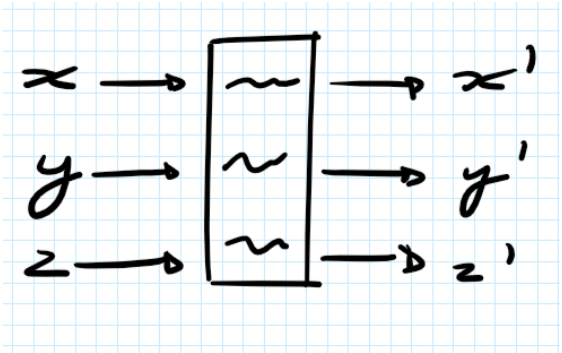


Figure 4.4: The most basic operation of a node in the graph, *bus* takes each substream in, processes it independently, then forwards it out on its own substream

Perhaps the most straight-forward operation, the *Bus* operation (see Figure 4.4) takes any number of input streams and maps them to an identical number of output streams - this has the effect of applying a unique instance of whichever user process is being used to each individual stream, where the data and environment is entirely isolated from each other. Note that this

does not inhibit the processes from communicating over other means (database connections, sockets, etc.) but merely ensures that they are logically isolated for the purposes of the data flow.

This is particularly useful for cases where the process being used is sensitive to continuous data streams, and would produce erroneous data if the input stream data was to be interleaved. By contrast, running multiple streams into a single process would result in interleaved data from each (with the order based on their arrival time), which while fine for processes that operating on discrete data, in cases where the process in question requires continuous data the output would likely be meaningless.

4.5.2 Map

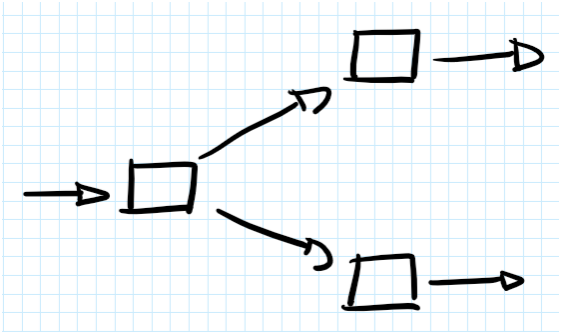


Figure 4.5: The *map* operation takes a single stream input, and applies each discrete message to its own processing function, before emitting each along their own substream; new streams are created on demand until there are enough to handle the function processing delay and packet input rate

The *Map* operation (see Figure 4.5) takes a single input stream, and attempts to take each discrete message received and apply it to an isolated processing function, creating new instances as required to accommodate the input rate. *Map* expects that the processing function emits messages at unity with the messages sent to it - one-in, one-out - this is required to allow *Map* to determine if the function is overloaded, and if more should be instantiated to compensate. The output data remains on a unique stream, (unmangled), allowing subsequent processing on separate isolated streams, although the precise stream-to-process mapping and longevity of each stream is not guaranteed, as the *Map* function may elect to set-up, tear-down or re-use any given processing function.

As this operation requires information about the process it is forwarding to (now to start, stop and detect completion) the *Map* operation will always be created as a node in the graph. If it were to be created in the network, without a proper node, it would lack the context to be able to perform the operation correctly - lack of a target instance-type would render this impossible.

For practical reasons, the upper and lower bound for the number of possible instances will

likely have to be specified in software, otherwise high throughput data going to long-running processes would quickly result in very high loads as many processes would be created to deal with the stream data.

4.5.3 Reduce

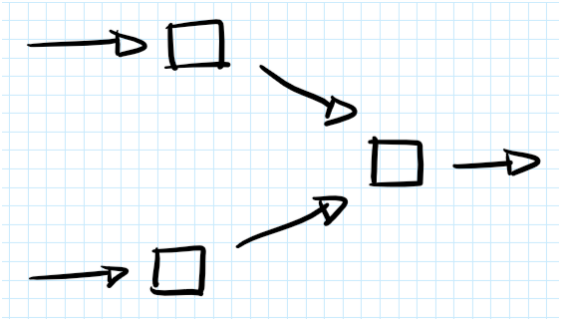


Figure 4.6: Performing the complimentary operation to *map*, *reduce* takes any valid substream and applies it to a single processing function in the order the packets are received, emitting all resultant data on a single stream (with the nodes address as its' source)

Reduce (see Figure 4.6) performs the logical opposite, complementary operation of *Map*, taking multiple streams and applying them to a single processing function, preserving the routing data with each packet (so each flow remains 'unmangled')

4.5.4 Mux (or 'Multiplex')

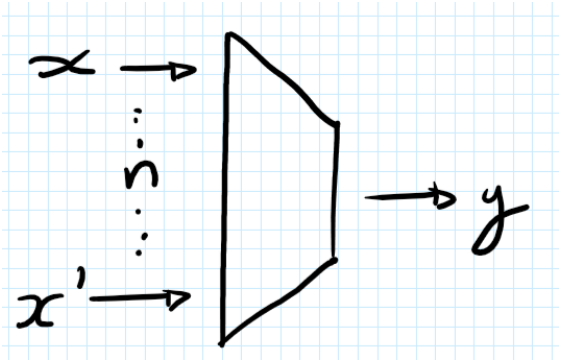


Figure 4.7: The *mux* or '*multiplex*' operation performs a variant of the *reduce* operation, taking any number of (valid) substreams and inserting them into a single output stream in the order they are received.

Multiplex (*Mux*) operations perform the opposite of Demultiplex (*DeMux*), taking messages from a single stream and dividing it over a number of outputs according to a distribution policy. Like *DeMux*, this operation explicitly mangles the messages received, and as such also must treat data as discrete packets, as otherwise the distribution would make no sense without

additional data; sequence numbers, for example, might be used by applications to track the 'real' order of the packets, despite their re-ordering mid stream, but are beyond the scope and control of this particular design, and should only be present in a higher-level protocol.

4.5.5 DeMux (or 'Demultiplex')

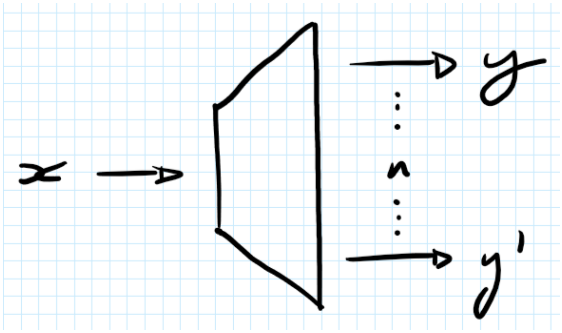


Figure 4.8: The *DeMux* or '*Demultiplex*' operation is a variant of the *map* operation, whereby a single stream's packets are distributed over a number of output streams, as defined by the node configuration and forwarding policy.

The demultiplex or *DeMux* operation takes disparate input streams and attempts to push them all to a single output. This is distinct from the *Map* operation, as whereas map preserves the address data with the messages, the *DeMux* operation explicitly *mangles* the destination addresses of packets received, replacing them with a pre-set target address.

This has the function of making it appear (to the target process) that the messages all came from the same source - a useful case for where the process does not want to differentiate streams, or is unable to handle multiple streams directly. Naturally, this will only work for *discrete* messages, as the order that packets are received by the target will be highly dependant on the order that they are received at the *DeMux*.

This is one of two operations that make particular sense to also support in the network infrastructure itself, without the requirement for an actual node.

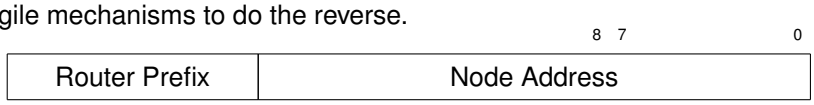
4.6 Address Range

The Linux kernel identifies processes by a Process IDentifier (PID) which is really just an alias for an unsigned integer with a device-dependant bit-width. Normal Linux installations will use 15-bits by default, resulting in an upper bound of 32,768 PIDs; this figure, however can be tuned, and the actual upper bound can be set as high as 22-bits, or 4,194,304 PIDs.

As both of these figures fit within a 32-bit unsigned integer with space to spare, we can, in theory, map addresses for nodes in the graph to their actual PID in the system - this would be especially useful should the user need to find which process was which node. This does fall

down, however, when sub-streams for wrapped processes are introduced, as they often run as multiple instances of an inner binary, so knowing the PID of the node maps to the process PID of the node only gets you the wrapper, not the actual process.

In theory, it would be possible to set the sub-streams PID to match the GraphIPC expected values, but doing so would require modification to the underlying kernel, or arbitrarily moving the node addresses around to match. It is a far simpler solution to use the kernel-assigned PID as part of the node address, and maintain the reference that way without resorting to other, more fragile mechanisms to do the reverse.



Unfortunately, this addressing scheme breaks down for cases where a single process (one aware of the GraphIPC network) exposes multiple streams, and thus addresses. In this case the source node address would be forced to be all the same if it were mapped to the PID of the process itself, limiting the possibilities for multiple output streams, so while convenient in some places, this design may ultimately hinder the network.

For this reason alone, a basic sequential numbering scheme is instead used with an 8-bit shift offset for the graph processes, such that they each have access to the bottom 8-bits for sub-stream addresses.

4.6.1 CIDR Compatible Notation

For both cases of address schemes, the 32-bit addressing conveniently aligns with CIDR dotted-quad notation. This is useful from both an 'understanding' and 'logical' point of view, as it allows for easy expression of graph node addresses, and aligns with the 1-byte reserved section of each node address for sub-streams, as shown in Figure 4.9.

Figure 4.9: A suggested address component allocation, although not required for the design to work, having an agreed upon format and scheme for addressing does greatly simplify understanding what the graph is doing.

4.7 Summary

To reiterate the beginning of this chapter, there appears to be a gap in the current methods available for communication between processes in configurations outside the traditional linear interconnections. This missing feature of inter-process communication systems is present

throughout all levels in the system, spanning as far as networking hardware, where it appears as capabilities of the routing hardware, external to the local machine.

With the increased complexity of the local run-time environment both for server applications and workstation use cases, there is a pressing need to address this gap, as the capabilities of the larger-scale networks are increasingly useful in the micro scale.

Taking cues from the macro-scale networks in production as well as the local interconnection schemes seen in various operating systems designs, this new inter-process communication design takes advantage of their general schemes, while also attempting to minimise the overheads inherent in re-implementing the full gamut of capabilities available in macro-scale networking equipment - most of which is of limited use at the micro-scale and would in effect simply reduce the overall throughput for the inter-network as a whole.

The implementation as described in the following chapter is a user-process level implementation, allowing the entire framework to be executed in user-space on a modern (kernel version 4.16.6 or newer) Linux-based system, but the design presented here could be translated down an abstraction layer to a kernel module with relative ease. The sole reason this has not yet been done (as of the time of writing) is that the development cycle in doing so would be dramatically slowed, limiting the exploration and testing of the overall scheme.

Footnotes

[36]The full POSIX API http://pubs.opengroup.org/onlinepubs/9699919799/toc.htm

[37]System-V IPC API http://www.tldp.org/LDP/lpg/node21.html

[38]MultiProtocol Label Switching, the working group for which can be found at https://tools.ietf.org/wg/mpls/, tags packets with 'labels' to identify stream or routing semantics.

Chapter 5

Implementation

At the core, GraphIPC takes a number of *Nodes* (each, a graph-aware process) and connects them together using a *Router* to marshal the messages between each *Node*. Thus; from the perspective of any given *Node*, the network appears to be a peer-to-peer form graph, with each node connecting to each other directly, whereas in reality, each *Node* simply forwards to their host *Router* (a graph-route-aware process to which nodes connect) which then connects the dots between them.

One particular oddity of this implementation is that the network operates more like an Multiprotocol Label Switching (MPLS) system, where in this case the source address is the label. the only components that know the routes that messages should take is the router, with nodes simply emitting data tagged with their own address, rather than the destination.

This, while odd by the standards of today's networked systems, does have significant advantages both in terms of limiting complexity at the nodes, and distributing workload somewhat. With the lookup operations on the router defining the flow of the data - performing a lookup actually provides the process with the context to forward the data, rather than performing any form of mask/matching as per 'normal' IP-based network traffic (for example).

This is not to say that the nodes in the network *cannot* know where the data is going, quite the opposite, as the nodes themselves are able to request that routes be added, removed or otherwise altered (priority, policy, etc.), but that they do not *by default* need to know how their data is being handled.

5.1 Overall Architecture

The overall architecture of GraphIPC is similar to the normal message passing IPC methods in traditional Linux operating systems. The illusion that GraphIPC presents is that each process is communicating directly with the target without interference, but the reality for this is quite different due to practical concerns of the design.

As in the regular IPC methods used on Linux systems (Both POSIX and System-V) the client processes connect to one another through a routing layer, a function which would normally be hidden inside the Linux kernel, but in this case, the router itself runs as a user space process as well.

From an architectural point of view, this is interesting for two reasons; firstly it means that as the router can be implemented as a user-privilege process, other router implementations are possible with specific goals in mind, and secondly, combinations of interacting routers are possible with varying privilege levels, as required. This effectively leads to a form of pseudo-graph-namespacing, whereby which router the process connects to dictates its ability to communicate with other processes (via mediating the routing stages).

In future developments, it is the intention to move the router down into the kernel as a full module, and have it communicate with clients via automatically generated Berkeley (file) Sockets, but its operation in either case will be the same; the only advantage to being a full kernel module is a reduction in kernel boundary crossings, which on a CPU affected by Meltdown and its subsequent patch should make a significant performance difference.

Even so, with the additional copy and buffering operations inherent in an approach based solely on user processes, the maximum throughput for any given NLP tool is generally quite low. Concordance line generating tools are likely to have the highest throughput, as they perform a simple sliding window at the word level to generate lists of concordances, but even in this case, the 'wire speed'[39] from GraphIPC is so much higher (See section 5.3.2.1 for the experimental enumerations for this) that the losses involved in the additional copying and transfer of data are rendered irrelevant overall.

5.2 The 'Router'

The router acts as a kind of IPC switching and routing device - each message is examined for the source address, then the corresponding target is determined before the packet is sent to that address.

The connections to the router do not map directly to nodes in the graph, rather they expose a kind of 'physical connection' style mechanism, whereby processes can connect to the router to configure it, but not participate in the data flow in the graph (or subgraph) itself. Routers can also be connected to one another, each holding an internal representation of their *subgraph zone*, along with routes out to remote (connected) routers; naturally, in this case, the routers themselves are concerned with the data flow along the graph arcs, but they themselves do not participate in the data flow, simply shuttling information around.

5.2.1 Routers All The Way Down

As a router holds a copy of the local subgraph, as with macro-network routers, this means that routers can also interconnect with one another. This is interesting for a number of reasons; one, as previously described, is that the routers can be used for a form of graph-namespacing, whereby only subgraph-local processes can route data to particular addresses (others in the local subgraph).

In Figure 5.1, this is graphically demonstrated with a pair of routers connected to one another, each handling a subgraph of the overall graph network. Note that although only a single link is shown transitioning between the two zones in this particular graph, the network could quite easily have multiple paths to get through to the second router (a graph with higher connectivity would demonstrate this). In actuality, the connection between the two router processes would be a single link, as there would be no need to connect multiple times to each other, but the virtual paths overlaid on that single link could easily be numerous.

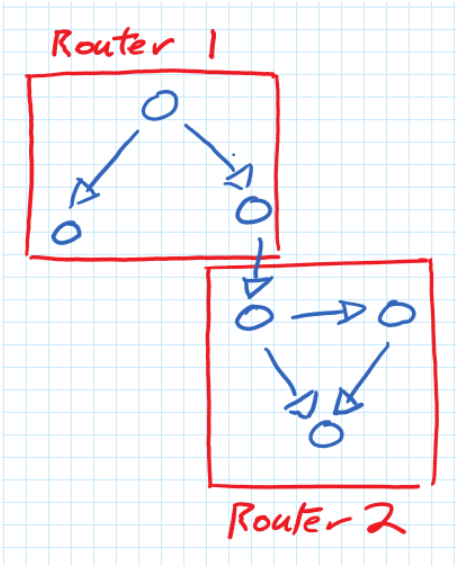


Figure 5.1: Two separate routers acting as a single graph. Each independently controls their *zone*, which can be modelled as a single large-address-space node for the purposes of forwarding and addressing.

The implementation presented here is written in plain C (compiled on gcc 7.3.0), as it is the intention that beyond the scope of this initial proof of concept instance, the router process would be brought down to that of a kernel module, and the user-space libraries would be implemented directly as system calls. Furthermore, by sticking to C99-compliant standard C, it is fairly straight-forward to implement wrappers for other languages, as the interfaces to C are extremely mature and expressive. To enable a simpler, faster development cycle, the router code has been implemented as a user-space process - this allows the full suite of development and debugging tools to be used to construct the binaries in a safer, more robust manner. This

also allowed development to continue without requiring significant virtual machine setup or host kernel alterations (potentially resulting in an unstable host system, while work continued).

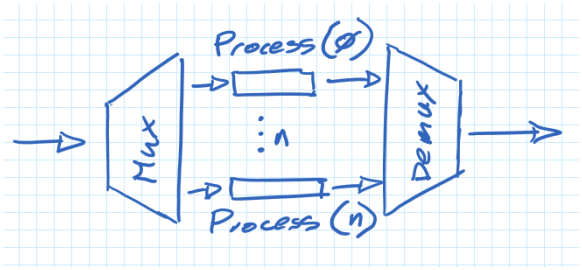


Figure 5.2: The internal (high-level) architecture of the `Graph` binary. The inner processes are wrapped binaries which can run unaware of the GraphIPC network

5.2.2 'Nodes' and Binary Wrappers

Nodes in the network represent individual endpoints to and from which streams can be routed. The illusion this gives is that the processes are directly communicating with each other, when they are in fact, communicating via their resident router, which determines where the stream data should be forwarded to.

As the nodes do not actually know which destination node their data should be directed to, stream packet headers only contain a source address (the node's own address) and the router itself fills in the destination when it receives the message, based on the declared links and forwarding policy.

To avoid the need to re-engineer each program to use the new IPC mechanisms that GraphIPC provides, it would be useful to *wrap* any process in a GraphIPC aware subshell-like environment, from which communications with other GraphIPC processes would be possible. As many Linux/UNIX processes use standard in and standard out (stdin/stdout) as their primary source and sink for data, it was possible to capture these streams and hand the data off to the graph without any modification required to the programs themselves. So in addition to the basic standard input and output bridge modes available with the prototype `Graph` binary, it is also able to fully enclose a subprocess, and with this modifies the inner processes' environment to capture the standard in and output streams, creating a graph-unaware, but compatable process.

Naturally, if the processes themselves wish to take advantage of the availability of a GraphIPC network, the process would have to link the libraries directly, but this allows for rapid integration of other binaries. This is especially useful for the initial and final nodes in a graph, as often the data sources and sinks beyond the GraphIPC network have their own driver programs, which rather than having to be rewritten, can be used as intended by their developers, but with a source or sink from the graph network. The simplest form of this is the use of `cat`, `echo`, `head`,

`tail`[40] and similar to input or output data from the network.

5.2.2.1 `Graph` or `Graph --bus`

As the network supports more than simple single-path communications, there are some concessions to be made for this to operate without mangling the streams. The default mode for the `Graph` binary is the `--bus` operating mode; in which the wrapper will spawn as many processes (on demand) as it has unique input streams, and will output as many output streams as it has sub-processes. This is a design form identified in section 4.4.

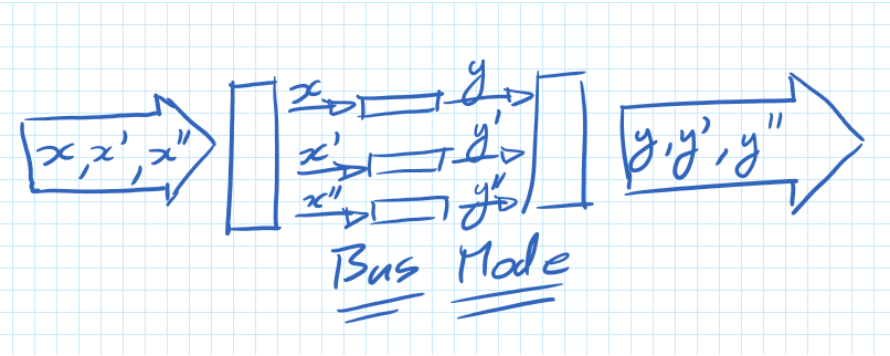


Figure 5.3: Graph in Bus Mode. Each unique input stream maps to a unique, corresponding output stream, such that individual data sources do not intermingle or interleave.

This is the most common operating mode, as a many wrapped processes will be unable to handle data from multiple sources, and the only way to ensure data stream integrity is to completely isolate one stream from another. By creating a new process for each unique stream ID on the input of the `--bus` option does tend to create rather heavyweight processing on the host running the graph, as entire, independent processes are created for even the smallest amount of data sent to a new stream address, so while it is possible to execute in this manner, a more graph-aware process performing this kind of operation would be preferable; ideally, one which handles longer packets as work units, otherwise there is the risk that *each byte* is handled as a new job, and thus a new process is spawned.

Furthermore, this is the easiest mode to understand, as it has no side effects involved with merging or splitting stream data in any way, but also conversely is the least graph-like operation, as the replication of processes could be achieved through the traditional Linux pipe mechanism, as demonstrated in Figure 5.4.

5.2.2.2 `Graph --map`

Whereas the `--bus` operating mode performs no stream mangling whatsoever, `--map` (explored in section 4.5) by contrast, takes each message pushed to the single node address - irrespective of source address - and runs each message in its own unique process, launching the process as

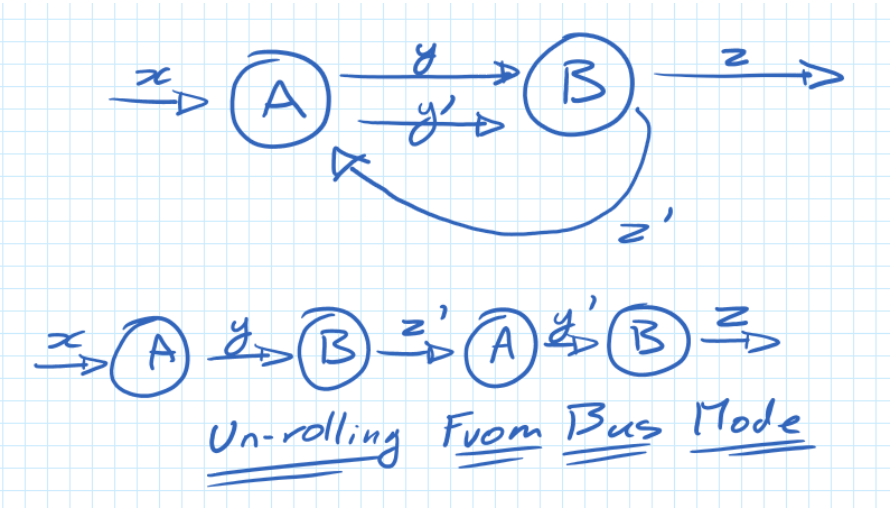


Figure 5.4: Bus mode vs. equivalent Linux Pipes

required as depicted in Figure 5.5. This can be an extremely expensive operation to perform for sub-processes which have significant memory, processor or start-up time overheads, as each time a message arrives, the entire setup-process-teardown cycle will need to occur.

To combat this for sub-processes where each input is treated independently anyway (where the data is entirely independent between transactions) the `--map` mode can be given the `--persistent` flag, which will cause the node to re-use any sub-process that has completed a transaction (transactions in this case are defined as one-in, one-out with regard to packets, so any process not conforming to this will cause problems) only spawning new sub-processes if the volume of input data exceeds what can be processed with the existing sub-processes, up to a limit as defined in the configuration or 255 sub-processes, whichever is reached first.

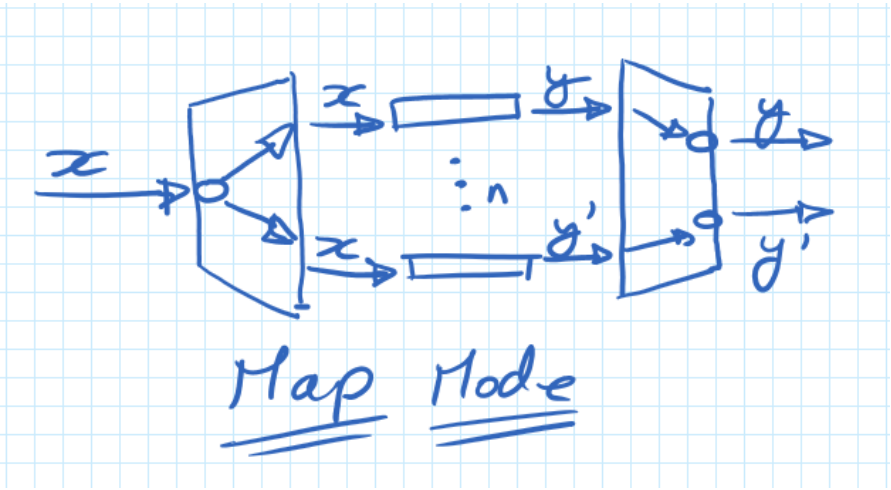


Figure 5.5: Graph in Map Mode.

5.2.2.3 `Graph --reduce`

While $--map$ ignores the input stream identifiers, regarding anything on its input as a single stream and processing each packet transaction separately, $--reduce$ performs the complementary operation, essentially doing the reverse.

$--reduce$ operations take each input stream, and spawn a sub-process to handle each unique input stream, then whenever a sub-process emits any data, the streams are combined in to a single output stream, appearing to come from a single address - namely, the node's own address. The model for this was discussed in section 4.6.

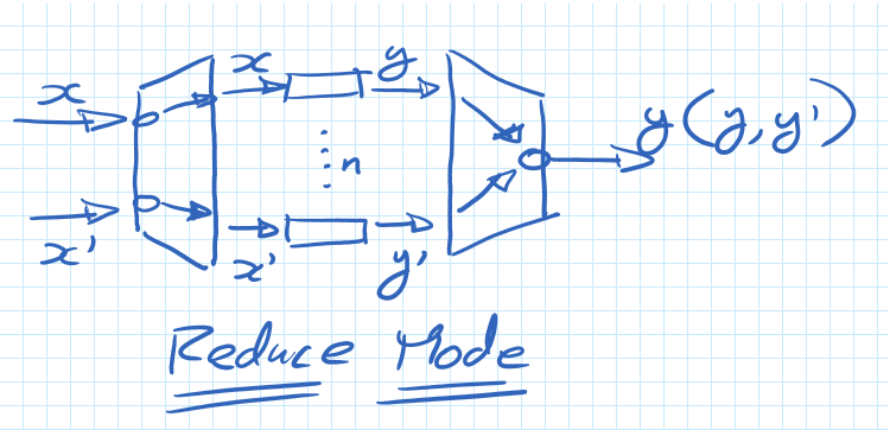


Figure 5.6: Graph in Reduce Mode

This node operation is mainly useful for any application which includes a final *collation* phase of any kind, taking the data and combining it into a single output for storage, or further processing where knowing the path is no longer important.

Note that `--reduce` is subject to the same sub-process count limits as `--map` but each node can be independently configured from one another.

5.2.2.4 `Graph --mux` and `Graph --demux`

During the development of `--map` and `--reduce` it became clear that there is sometimes the need to perform similar stream operations without there actually being a processing step - simply discarding the stream identifiers at one side or the other of a no-op operation.

To this end, the `--mux` and `--demux` (see sections 4.7 and 4.8) operations were created to fulfil this niche; `--mux` or *multiplex*, takes a single stream input on the node's address and emits it to any connected nodes in a manner dictated by the output operating mode. `--demux` also known as the *demultiplex* operation, does the reverse, accepting any sub-stream message as an input, combining them to a single address (the node's) in the order that it receives them.

Both are relatively simple 'utility' functions, but they can make a real difference both in terms of actual performance by cutting out the need for additional sub-processes for simple operations,

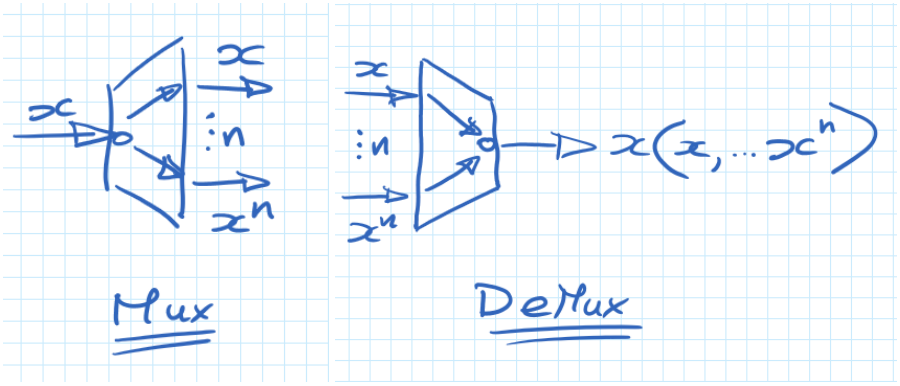


Figure 5.7: Graph in Mux and DemuxMode

and for situations where the graph reaches higher complexities, as the operations are easily understandable for workload designers.

5.3 Networking

Fundamentally, GraphIPC presents a kind of local-only network through which processes communicate in packets; so, rather than attempting to mask this, the discussion here will simply treat the network as such, and describe its implementation in networking terms.

5.3.1 Asynchronous Messaging

By explicitly declaring that the messages in this network are asynchronous, we avoid several problems. Firstly; if synchronous messages were possible, processes involved in cyclic networks could potentially lock up completely as the ring of processes all wait for a response from each other forever.

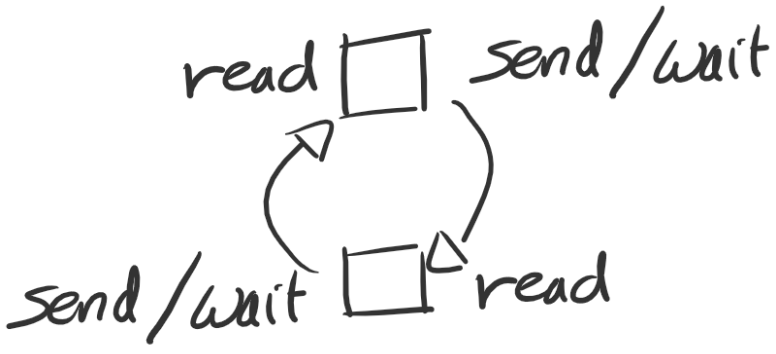


Figure 5.8: Cyclic graphs can cause deadlocks if messages are synchronous

Detecting this kind of deadlock can be tricky, especially if the messaging is not all using the same method to communicate - a scenario involving UNIX pipes as well as GraphIPC would

break any subgraph modelling designed to avoid the deadlock scenario, as the UNIX pipes would be effectively invisible to the subgraph analysis stage.

Secondly; even in acyclic graphs there would be form of 'event ripple' for every input message, locking subsequent processes into lock-step with the slowest process. Assuming an initial packet source that can supply messages as fast as a downstream consumer can process them, at each stage, the slowest process would simultaneously halt all preceding message transfers as the channel backs up, and any subsequent process would be starved waiting for any messages from the slow one.

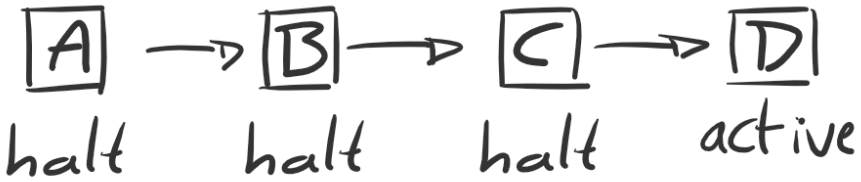


Figure 5.9: Synchronous messages can cause ripples of process-halts to propagate over the network, as each waits for the next in sequence.

While we cannot prevent the message starvation deeper in the graph (although parallel processing and process duplication do address this somewhat), provided that the initial packet source is not infinite, we can exhaust the source and remove it from active memory, leaving only the messages 'in-flight' to track. Doing so will likely have consequences relating to memory usage of the graph itself, but the savings on being able to tear down a process and it's associated resources may be useful in and of themselves.

5.3.2 Protocol

Figure 5.10 shows the structure of a graph network packet, and while the structure itself isn't mandated to be ordered, the listing here shows the actual order of the fields in the packet as well, with only the addition of a free-form, schemaless payload after the `length` field.

Serialisation and deserialisation are handled as part of a packet library written for this project which ensure that multi-byte fields have their endianness preserved across the connection to the router.

The `magic` field is always transmit first and is used both as a frame sentinel, but also as a fast way for any listening processes to discard a packet if the 'magic' does not match.

Also included is a `version` field, which should always be the same at both ends of any given connection. This is included mostly as a sanity check for during runtime to ensure that all binaries accessing the router are using the same version of the protocol.

```
1  /** GNW Packet header structure */
2  typedef struct {
3      uint8_t       magic;
4      uint8_t       version;
5      uint8_t       type;
6      gnw_address_t source;
7      uint32_t      length;
8  } gnw_header_t;
9
```

Figure 5.10: The gnw_header_t structure, which describes the header of all GraphIPC packets

`type` defines if the payload data is raw data to be processed, or if this is a network internal 'command' message.

Commands are used to handle address negotiation and to configure the running router instance for new connections, policy changes, and other parameters to control the graph network as a whole.

The `source` field is a 32-bit unsigned integer following the schema discussed in sections 4.6 and 4.6.1, and is used to identify the packet source address. Usually this will be an actual node address, but in some cases it may be a sub-address for a sub-process within a node.

By convention this will be indicated by the lower 12-bits being non-zero, as the default mask is `0x00000FFF` for node-local addresses.

The final field in the header is the `length` entry, which defines the length of the remaining packet data, with a maximum of $2^32 - 1$ bytes per payload. In most cases, this is actual data to be passed on to either a child process (in the case of Graph wrapping a traditional Linux process) or to be output by the node for further processing.

In the case of command messages, the data in these fields has special meaning for address negotiation and responses, as well as the various parameter altering commands.

5.3.2.1 Digraph Storage and Performance

There are two generalised forms for storing a digraph structure in memory, and both have their benefits and detractors for a given application; the edges can be embedded as a property of the nodes (or visa-versa, with edges embedding the nodes, although this arrangement is more rare), or as independent entity structures, one for nodes, one for edges.

5.3.3 Address Lookup Mechanisms

Initially, the approach chosen to handle address lookup in this implementation was to perform a byte-level lookup for each byte of the address, most-significant-byte first. If the address fully matches the stored one at this first stage, then the function can immediately return with the context (as demonstrated in Figure 5.11), otherwise it progresses one level further in and one byte deeper into the address - essentially performing a *shortest unique prefix first* lookup (as

shown in Figure 5.13).

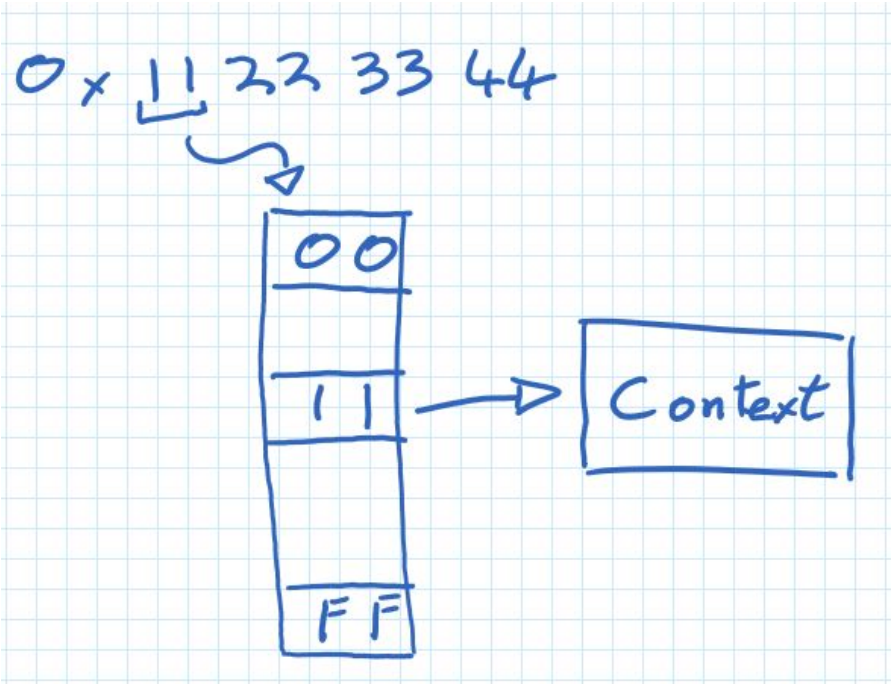


Figure 5.11: The fast address lookup path through the address data structures - the first level can reference the context for the handler immediately, short-cutting the entire structure, provided that the address field in the context entirely matches the needle address

This structure also allows for shorter masked address lookups to refer to the lowest next-level address in the structure, effectively providing a broadcast address which hits the first next-refined address in the subsequent set.

This is useful for this particular design, as we use the $/8$ mask to refer to sub-streams within a given node, leaving the $/16$ mask (the level above in the structure) to refer to the node that contains said streams. As a node will *always* request an address before any of its sub-streams, the shortest match will always be the *host node*, saving a lookup during the address resolution stage of handling a received packet. Note also that this approach does mean that offsets in each table are known and easy to compute (see Figure 5.12) such that each byte offset can be directly known without any searching required.

```
(uint8_t)((lookupAddress >> (8*(3-maskBytes))) & 0xff)
```

Figure 5.12: Computed offset in the local table, where 'maskBytes' is actually the offset in the mask lookup table, for speed.

Generating the masks to perform this lookup was done often enough to make a case for optimising the operation; to that end, an array of pre-baked bit patterns was used to reduce the generation time for each mask (See 5.1, resulting in some minor speedup overall.

```
1 const uint32_t mask_lookup_table[4] = {
2         0b11111111000000000000000000000000,
```

```
        0b11111111111111110000000000000000,
        0b11111111111111111111111100000000,
        0b11111111111111111111111111111111
};
```

Listing 5.1: Generated structures such as this mask lookup table are used to speed up the lookup operations

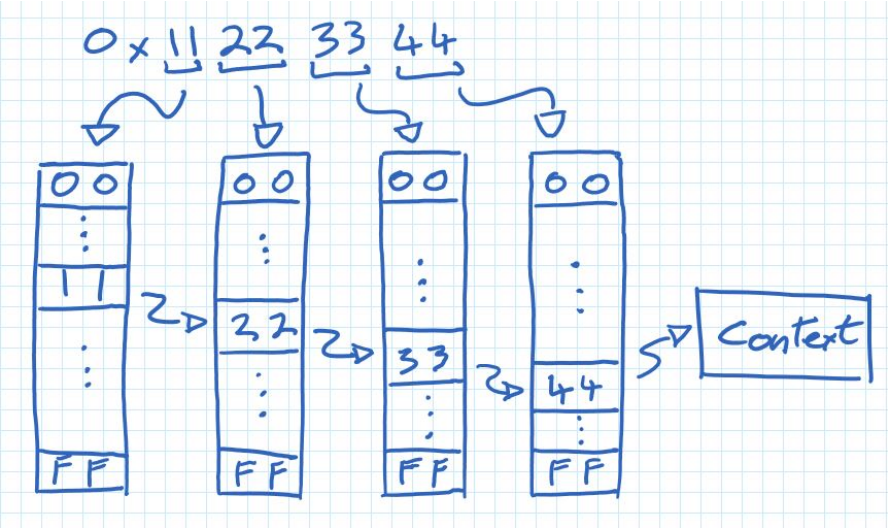


Figure 5.13: The slow address lookup path through the address data structures - this would only normally occur if there were many addresses with very similar values, as by the second level, the data is sufficiently different to specify the handler without ambiguity

Each level of the lookup is represented in memory as an array of 256 elements, one per possible byte value, which contain the current address mapping for this byte/mask combination, and if required a pointer to additional sub-tables which do the same for a finer-grained byte offset. As the individual tables are essentially plain in-order arrays, byte-level queries can be performed extremely quickly, which contributes greatly to this approaches performance. Further aiding the performance of this structure is that each table fits entirely within a memory page, reducing the number of page table loads required to do each lookup.

Naturally, the trade off here is that the arrays are largely empty for sparse input sets, so the overall structure is very space-inefficient. Some better memory efficiency could be achieved with the use of a pointer arrays, rather than storing the state of each value in the array itself, but in doing so it would introduce another indirect lookup to every level, which may adversely affect performance in and of itself.

Overall, this structure achieves, an average ≈ 230 nanoseconds per address lookup - some are naturally much faster, requiring a single lookup operation, others are longer requiring up to 4 lookups. This means that barring any other factors, such as formatting the packet to be sent out, or other internal processing delays, the absolute upper bound for throughput lies at

slightly over $4,347,826$ lookups per second, which with an MTU of 1500 results in a theoretical maximum throughput of ≈ 6.074 GiB/s ($\approx 6,521,739,000$ B/s).

Naturally, this is a theoretical upper bound, and any processing delay inherent in a real implementation of the router will cause this figure to drop, but as it sits so much higher than the actual maximum throughput of the loopback interface, we can say that the address lookup time is negligible with respect to the overall performance of the binary.

However, despite the successes of this structure and algorithm, it has one major drawback - storage space. The size of the structure quickly becomes unmanageable for large address sets, and as the lowest octet is used for substream addressing, performing lookups for anything other than very sparse address sets quickly becomes *at least* a 3 stage process, if not a full-depth 4 stage lookup. At this point, the structure becomes no better than a balanced tree approach, using the table model for graph storage.

Attempting to replicate in software what would be normally handled in particularly high performance hardware in network routers - which as has been shown so far, can quickly be doomed to failure by the memory requirements. Instead a more pragmatic approach was ultimately taken where the addresses are stored in a hash table structure; a balanced search tree implementation, with the intent to minimise the lookup overhead.

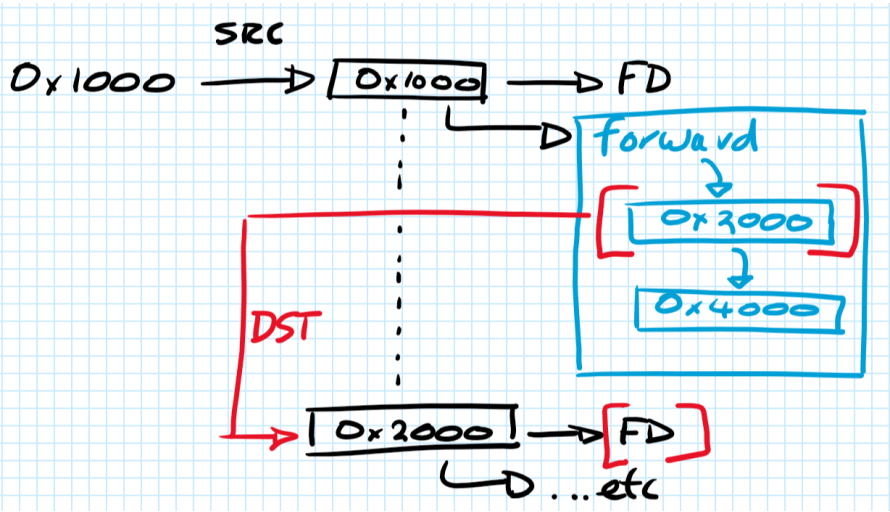


Figure 5.14: The hash address lookup sequence implemented in the GraphRouter binary.

As shown in Figure 5.14, this structure may result in a slightly slower average case lookup time, but will remain much more consistent when scaled up to large numbers of addresses. This prevents the router from experiencing unpredictable *jitter* as lookups take unknown amounts of time to complete.

```
/**
 * Context for a given connection to a running node or subgraph-router.
 */
typedef struct {
    kvec_t( gnw_address_t ) forward;

    int forward_policy;
    int bound_fd;

    int state;
    uint64_t packets_in;
    uint64_t packets_out;
    uint64_t bytes_in;
    uint64_t bytes_out;
} context_t;

```

Figure 5.15: The content of a context structure for tracking the connection and data for a single graph node

5.3.4 Forwarding and Policies

As this design requires that the router hold information about the network - rather than just a set of known addresses - the routing software needs to be able to do both fast address lookups *and* fast route lookups when messages come in. The messages themselves only contain the *source* address, so cannot be forwarded in isolation. This means that each graph node in the network needs to have a *context* held in the router containing all this required information.

The structure for this context data is shown in Listing 5.15; and beyond some amount of statistics storage (the various packet and byte counts) the structure only tracks the current forward policy, the file descriptor of the connection to the node this structure represents (although this may be NULL for one which does not currently exist) and the forward list, stored as a vector.

The node itself, as represented by the Graph instance follows the general scheme shown in Figure 5.16 for all messages received, with exceptions for when running as a bridge to normal Linux standard input or output streams. This sequence has the messages demultiplexed, processed in the sub-process the multiplexed on before being output to the connected GraphRouter instance.

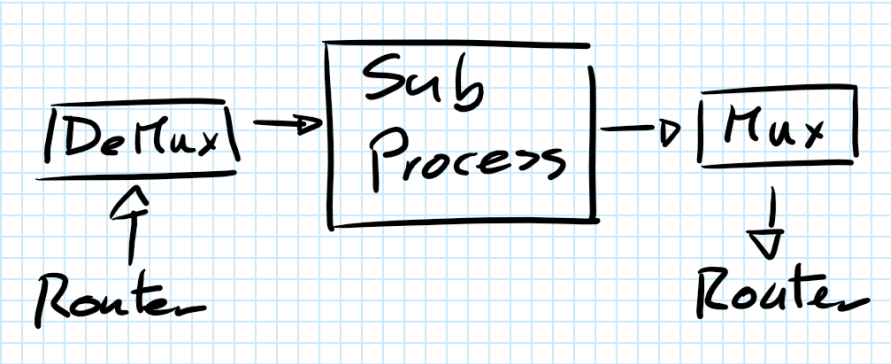


Figure 5.16: Subprocess Wrapping

Multiplexing simply works on source address addresses, and either operates as a `noop` mode, which does nothing to the address before forwarding it, or alternatively it can operate

in a `merge` mode whereby messages from multiple inner processes (if any) are masquerated as having come from the node address itself. This is done in cases where is not desirable for multiple processes to be invoked at the receiver, or where the precise source for a given message is not required (as would be the case for a load-balancing situation).

Demultiplexing can operate in one of three modes, the first is `noop`, whereby no alteration to the incoming source addresses is done before processing. In this mode, if an inner process is needed to be spawned, a single one will be created and all messages will be forwarded to it directly.

Its second mode, `merge` operates much the same as the `noop` mode, but in addition to spawning a single inner process, if required, it will also mangle the message source address before passing the data to the inner process.

In the third mode `spawn` the Graph binary will spawn a new inner process for each unique incoming source address and forward the message to the new inner process. If an existing process is already handling the source address, it will receive the new message without the Graph binary spawning another.

In combination, the multiplex and demultiplex stages provide a number of unique operating modes and give the user options for how to handle input and output data at each node, independently of the router itself. As a side benefit, this also takes some of the processing load from the Router, leaving it to handle address lookup and message forwarding alone.

In a kernel module implementation, the process context and socket ID can be tracked together in process states, rather than independently as it presented here. In practice, while the kernel module may involve more of the system, the actual implementation of processing the messages may be simplified as the kernel has direct access to the socket buffers in use, so can use them in-place rather than having to buffer independently.

The buffer itself is a fixed-length simple flat buffer implementation, which prevents the working set (in RAM) from growing during operation, and eliminates any allocation or deallocation delays which may occur due to the system implementation for heap management (malloc/free).

This forwarding lookup is structured quite differently from the address lookup, as the forwarding operations require that sequential fast access is maintained as often as possible. Consider round-robin routing policies, whereby each individual packet needs to be forwarded to a new endpoint - for this case, the list structure for the forwarding lookups is ideal, as the *next* cursor can be maintained between packets, only progressing when packets are forwarded. This approach results in a nearly O(1) lookup complexity.

5.4 Binaries

There are only two binaries for this project - excepting the unit testing framework - `Graph` and `GraphRouter`. GraphRouter initiates and controls a host's IPC router, while Graph creates, wraps and configures nodes in the graph.

The command line parameters and switches these support are available through `-h` and `--help` on either binary, but are reproduced and expanded upon here for context in the latter chapters. Note that many of the parameters have both a long and short form; when this is the case, both are specified in-line here.

5.4.1 `GraphRouter`

The userspace router for GraphIPC messaging

`--status`

Request a status message from a running router instance. This will attempt to connect to any running router instance and cause it to report a number of statistics on its current operations.

`--policy`

Change the link policy between a source and a target. This requires that `--source` and `--target` also be present on the command line to determine the arc in question. At the time of writing, the valid policies are 'broadcast', 'roundrobin' and 'anycast'.

`--connect -c`

Connect a source to a target, with the default (broadcast) policy. This requires that there by a `--source` and `--target` also present on the command line.

`--disconnect -d`

Disconnect a source from a target. This requires that there by a `--source` and `--target` also present on the command line.

`--source -s`

The source address of the arc to modify. Must be a valid node address.

`--target -t`

The target address of the arc or node to modify. Must be a valid node address.

`--mtu`

Force a particular MTU - setting this too high may cause excessive packet loss if your hardware is unable to handle the load. This will default to the loopback interface MTU on the host, or, if this is not available to be queried, will use 1500 bytes.

`--dot`

Output the graph in DOT format periodically, rather than status messages or the address table. This is mostly just useful for debugging, and renders the entire graph in DOT notation.

`-v`

Increase log verbosity, each instance increases the log level (Default: ERROR only). Must be called first to have effect on subsequent operations called by other parameters.

5.4.2 `Graph`

Wrap a normal Linux process stdin/stdout pipes with GraphIPC connections to a GraphRouter process Allows non-compliant programs to be used in a graph

`-h --host [host address]` **and** `-p --port [port]`

Defines the GraphRouter host address and port. When multiple routers are run locally (not recommended, as this splits the local graph) this allows the Graph process to select which router to use. By default this will use the first local router it finds. The current prototype only supports single-homed routers, where there is a single router on each host, each controlling the entire local zone. If multiple routers are running on the same host, each will have an entirely independent address pool, with no communication between the two; quickly causing duplicate addresses to apparently exist.

`-a --address [hex address]`

The (requested) hexadecimal node address, may not be respected by the router. Cannot be 0. Routers will attempt to satisfy this request, but if the address is already in use, another address will be allocated automatically. The only other alternative to this behaviour which would not negatively affect the current graph would be to simply reject the request and have the client process quit, but for simplicity the current, fallback behaviour is used.

`--immediate`

Start running the inner binary immediately. By default wrapped processes are only started on demand when data arrives

`-v`

Increase verbosity, repeat for increasing levels of detail. This should be specified early on in the command line flags, as subsequent arguments may use the pre-existing value otherwise.

`--`

Optional separator between Graph arguments and the inner binary. While Graph will

attempt to assume parameters beyond the binary definition are intended for the wrapped binary, this argument *enforces* that any subsequent arguments are intended for the inner binary.

`-i --input`

Executes Graph in 'input' mode, where any UNIX pipe messages sent on the standard input stream are re-emitted by the Graph node.

`-o --output`

Executes Graph in 'output' mode, where any Graph messages are immediately written out to the standard output stream, with the GraphIPC protocol bytes removed.

`--mux`

Configures the multiplex stage of the Graph binary, and will only accept 'noop' and 'merge' as values.

`--demux`

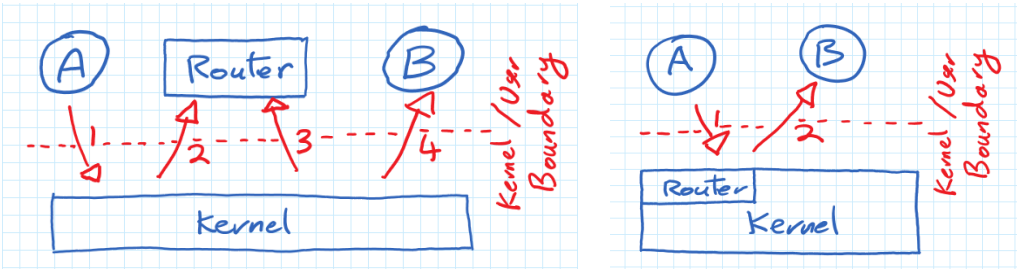
Configures the demultiplexing stage of the Graph binary, and will only accept 'noop', 'merge' and 'spawn' as values.

5.5 Summary and Future Developments

The implementation presented here consists entirely of user-space processes, which while useful for development, debugging and potentially (in production) deployment, does mean that there are technically superfluous memory copy operations going on during communications, as depicted in Figure 5.17a. The double kernel boundary crossing required by the sockets interface (twice for a node to send to the router, twice again for the router to forward the message) will have significant impacts on overall performance. The application space that the subsequent tests exist in is only able to avoid this performance drop impacting the overall throughput as the programs actually processing the data at each node have so much lower bandwidth than the GraphIPC network that the losses in the system itself vanish into the noise floor.

In continuing this project, the next logical step is to move the entirety of this router design down into a kernel module (See Figure 5.17b), removing one of the boundary crossings at each transfer (becoming a user-kernel-user sequence, rather than user-kernel-user-kernel-user sequence).

While lacking in some features, this example implementation of the GraphIPC design is easily able to demonstrate the interactions possible with the overall model. One of the major missing features is the ability to connect multiple routers together; this is largely due to the problems of address-scoping for allocating new node addresses in each router - by default the



(a) User processes (b) Kernel module

Figure 5.17: Graphical representation of the kernel boundary crossings involved in both the user-space and kernel-module implementations for the router component. Boundary crossings are generally handled through system calls in normal Linux configurations, although mapped memory and synchronisation primitives can also be used to achieve the same affect with some implied, unpredictable latency (unless executing as a RTOS kernel)

implementation here is only able to monotonically increment its next address - albeit by 256 each time, as to allow for 255 sub-streams per node - and has no mechanism for address negotiation within a collection of routers.

One possible solution in the simpler end of the scale would be to give the router itself a ‘zone address’ which would act like a DHCP range for that given router, as well as potentially having the router itself be modelled as a node - although doing so would have its own interesting issues with representation, as routers are supposed to be effectively invisible to the nodes running on them.

Footnotes

[39]The author notes that this isn't what 'wire speed' actually means, but for the purposes of analogy it is useful in this instance

[40]See http://man7.org/linux/man-pages/man1/cat.1.html, http://man7.org/linux/man-pages/man1/echo.1.html, http://man7.org/linux/man-pages/man1/head.1.html and http://man7.org/linux/man-pages/man1/tail.1.html respectively

Chapter 6

Evaluation

With the prototype implementation complete, the following sections detail a number of the tests performed to determine both the performance available with such a system composed entirely in userspace, in addition to describing the practicality of defining the more complex networks.

Initially however, some discussion of both the test environment (the machine configuration) and the test software (the tooling) is required. In the latter sections of this chapter this work turns to a more general evaluation of the system as a whole and how it fits in with other aspects of processing workflows.

Overall the results in this chapter provide a mixture of complementary evaluations: quantitative (described in sections 6.1 to 6.7), followed by qualitative and application based (described in section 6.8).

6.1 Test Machine Configuration

The full hardware and software configuration the following tests were performed on can be found in Table B.1, but in brief, the computer was an unmodified 2018 Lenovo ThinkPad T480s with an Intel Core i5 8th Gen CPU running Arch Linux[41] kernel version 5.2.11-arch1-1-ARCH 1 SMP PREEMPT with GNU gcc 9.1.0, and GNU make 4.2.1 with cmake 3.15.3 were used to control the builds. The entire project is also known to compile against Ubuntu Linux[42] via Travis-CI[43] builds, and although not tested in that environment, it should perform similarly, as the configuration described here is fairly unremarkable.

6.2 General Methodology

With the exception of the initial baseline throughput tests (see section 6.3) the tests in this section follow this general configuration:

1. Start an instance of the router (GraphRouter binary)

2. Configure all forward paths and policies
3. Connect all output and monitor processes
4. Work backwards creating any intermediate processes towards each data source
5. Instantiate the data source, beginning the test.

Statistics on throughput, packet counts and several other features are collected at relevant points throughout either via the graph network itself, or via tools such as pv (See section 6.4). This test cycle is then repeated changing a single parameter each time; common altered parameters are the packet size and the number of packets to process.

To replicate any of the tests demonstrated here, see the GraphIPC/tests directory in the project git repository[44]; each test is orchestrated from its own bash script, so can be executed on any machine with the requisite tooling. Additional scripts include aggregation tools to build processable CSV files for graphing and further analysis.

6.3 Throughput Limits

Internally, the GraphIPC prototype uses local sockets to establish the communication channels between individual parts, therefore the absolute maximum throughput of the system is governed by the maximum throughput possible for a local socket. In the specific case of the prototype, this local socket is bridging two user-space processes, whereas in a kernel module based router implementation, this would be between the Linux kernel and the graph-enabled processes.

6.4 Custom Tooling

To perform the tests described here a simple binary (ArgTest) was constructed to simulate generating messages, as well as simulate processing messages mid-stream in predictable, configurable ways.

6.4.1 `ArgTest`

To have a controlled environment for testing the performance and behaviour of the prototype, an additional binary was created to simulate the behaviour of external processes in a predictable way.

ArgTest is a very simple application which can perform the following operations:

- Generate messages of a fixed size with an arbitrary message delimiter.
- Emit messages at a fixed, known rate, with counting limits.

- Emit messages when sent a message itself, optionally with a fixed delay.

All of these features are configurable from the command line, through the s,d,i,c and w command flags, the definitions for which are presented here for clarity in the remainder of the chapter where commands are stated:

`-s [length]`

Generate a $length$ number of $'?'$ characters as the message payload, these will be sent with the addition of the delimiter string, such that the final payload will be $length+delimiterLength$ long.

`-d [string]`

Specify what the program should use as its delimiter string - this is only actually used for emitted messages, whereas all incoming messages use a fixed newline $''$

`-i [interval]`

Define the interval either between messages, when free-running, or the delay between receiving a message and emitting one ourselves if waiting for input. The value is in microseconds ($usec$, or one 1-millionth of a second), although the exact precision of this figure is determined by the kernel itself, as it relies on the system clock being accurate enough to represent microsecond ticks. If this is zero, no delay is applied.

`-c [count]`

Sets the number of messages to be emitted from this process. To prevent run-away processes, there is always a hard limit on the number of messages a single ArgTest instance can emit, although as this figure is defined as an unsigned long, in practical terms, the limit is beyond any required for testing (4,294,967,295 messages on a 64-bit system).

`-w`

Wait for standard input to trigger emitting a message after $interval$ delay. Input is buffered, so if the input rate exceeds the output rate, all will eventually be sent, after the cumulative delays expire.

6.4.2 pv from pv4science

As part of the evaluation of various programs here, there came the need to monitor features of a regular Linux style pipe; the 'normal' tool for doing this would be pv which can be inserted in a pipe-chain and will periodically output a number of statistics about the data flowing through at that point.

However, pv is a tool designed for human-readable output, so automatically converts (with significant loss of precision) to SI unit equivalent values (1400 bytes/sec becomes “1.4k/s” in its output). To prevent this, I modified pv to inhibit this behaviour and have it return the raw, actual long-double floating point values it internally uses, thus allowing for higher precision of measurement.

```
15017.000000 0:00:01 15016.444392 15016.444392
31131.000000 0:00:02 16114.306172 15565.359912
46731.000000 0:00:03 15599.766004 15576.828655
61664.000000 0:00:04 14932.940268 15415.857403
```

Figure 6.1: An example of the modified output from pv in the pv4science package; this particular example shows, in order, Number of messages, Timestamp, Throughput rate in bytes per second, and average throughput in bytes per second.

The code for this modified pv is available on GitHub at https://github.com/JohnVidler/pv4science should this minor modification be useful to any other developer.

6.5 Standard Linux Pipes

To provide a baseline characterisation to both describe the test system and for comparing GraphIPC, a simple test was run to transfer a fixed number of messages with an increasing payload size using ArgTest. Taking these measurements and plotting them against one another we see the a mostly predictable behaviour (See Figure 6.2).

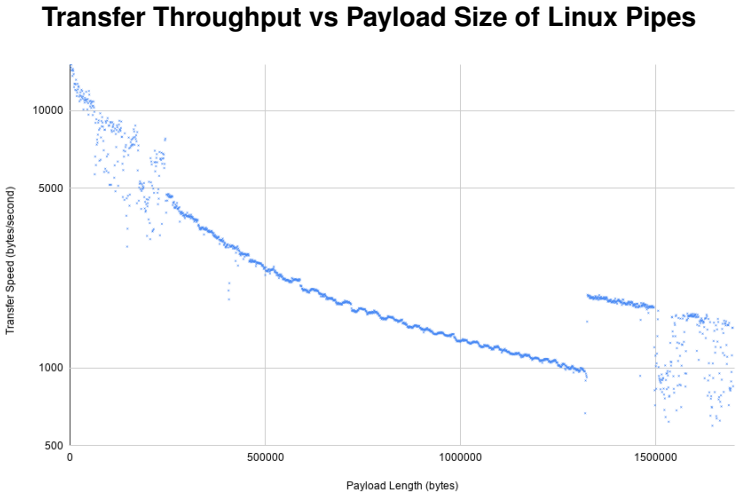


Figure 6.2: A plot of the effective throughput for unix pipes. The low-end noise is likely measurement error from the extremely small run duration for those tests. The upper ‘step’ is likely to be an increase in internal buffer size or transfer approach causing the local increase in throughput. See also Table B.2.

Of note however, is that at the low-end of the payload size, the transfer speeds are so fast, and thus; the times taken to transfer the data so short that the accuracy of the measurement

drops off with the system's timer resolution. Therefore, while the data does generally follow the same curve, the precise values for payloads smaller than approximately 250,000 bytes (250 kB) cannot be trusted completely.

This is a limitation of this kind of instrumentation, and is present in all subsequent tests in the chapter, unfortunately. To mitigate some of this problem, an intentional 1 second delay was added before each test is started but while pv is already executing to allow the timers to stabilise, and payload sizes were chosen to start very close to the top of uncertain range.

Of further note is the sudden, marked increase in throughput at approximately the 1,320,000 byte-mark (1.32 MB) is extremely likely to be a doubling of the internal buffer capacity, as the increase in throughput is almost precisely doubled (an increase of approximately 956.24 bytes/second once stable).

6.6 Local Sockets

As the prototype uses local sockets as a transport layer for messages, rather than a packet based message passing interface to the kernel, there is a hard limit to the performance available to the prototype, as defined by the maximum throughput capable on the test machine for TCP sockets.

At the time of writing, no general characterisation of TCP sockets on loopback interfaces could be found, so instead the following data was collected to characterise the link (barring some error caused by the measurement tooling) and plot in Figure 6.3.

Loopback TCP Socket Throughput with Increasing Payload

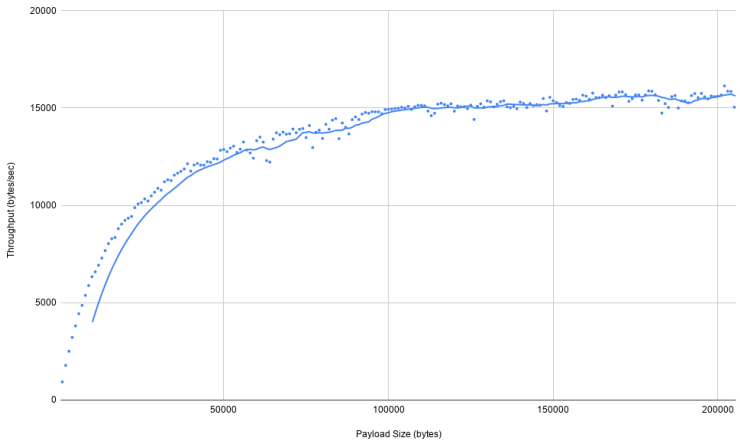


Figure 6.3: Throughput via a loopback TCP socket between two nc (netcat) instances on the same machine as all the following tests.

As both ends of the communication were monitored, the difference between these can also be plot against the payload size, rendering a view on *throughput loss* over the TCP link (see

Figure 6.4).

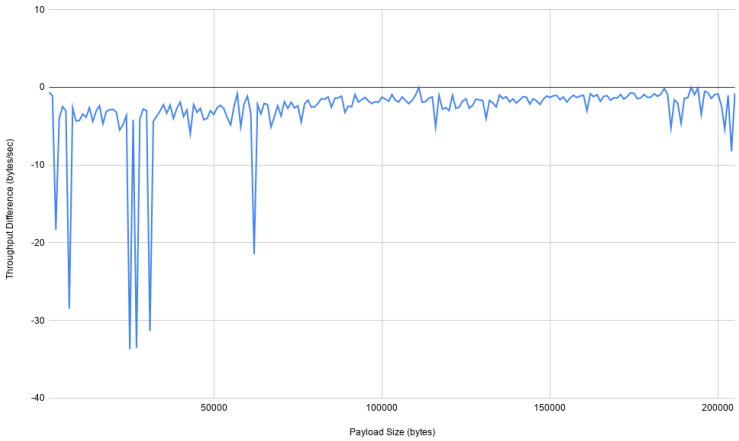


Figure 6.4: The difference in throughput between the transmitter (positive throughput) and the receiver (subtracted from this throughput). This results, as would be expected considering the transport overheads, in a negative value with a magnitude describing the total throughput loss. Note that the lower-end noise is from the inaccuracy of the measurement tooling.

Another limitation with the prototype worth mentioning before delving into the various performance aspects is the packet overhead inherent in the addressed format. In opposition to stream-based systems where the data transmitted is purely meaningful to the destination, GraphIPC also needs to transmit metadata about where the packet has come from such that the router is able to correctly direct the message. This in of itself causes an 11-byte overhead for all messages, such that packets of *less than 11 bytes* will be sending more metadata than actual data in the stream, effectively slowing the overall transfer down with reference to a purely data-oriented flow.

However, with this known, the actual upper bound on throughput is unlikely to be a problem, as the retrieval and processing tools usually used for natural language processing tend to be either quite heavyweight (written in scripting or interpreted languages, or requiring large libraries) or tend to take considerable time to perform the various statistical operations required. This speed limit imposed by the tooling will be much, much lower than the available processing speed here, so in practice, the dominant performance factor will be the tooling in use, rather than the transport.

6.7 Forwarding Policies

The GraphIPC router prototype supports three forwarding policies; broadcast, anycast and round-robin. Each of these policies has ramifications for the overall load on the router and its clients, and therefore, will have different effects on throughput performance.

Therefore to determine the characteristics of these different operating modes, each was tested in the same configuration; specifically a single data source with two data sinks - and the results examined to establish overall trends for each.

6.7.1 Broadcast

With the router in broadcast mode all messages are sent to every forward address (the default mode for forwarding messages). For each iteration, the router was shut down and reset, along with all connected nodes to ensure that the start state was identical in all cases. Each sample increased the payload length by 1000 bytes to cause the transport layer to send more than a single packet, as the loopback network MTU (maximum transmission unit) is particularly high - in this particular case 65536 Bytes (≈65KiB), so small transfers will be unlikely to see more than one packet transfer in practice. Real network links will have much reduced MTU sizes, often either 1500 Bytes or 1470 Bytes, depending on the specifics of the network hardware in use.

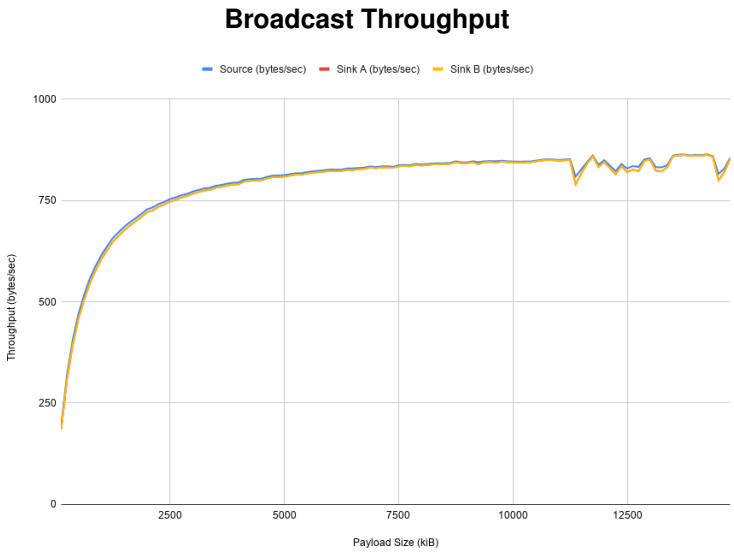


Figure 6.5: Source and Sink throughput with the router in Broadcast mode. From data in Table B.3

Transfer rates were monitored at both the source and sink nodes via the modified pv binary (See 6.4.2) then graphed against one another to produce Figure 6.5.

As would be expected, there was a little deviation between the throughput seen at the sinks from that at the source. This small deviation, a maximum of approximately 12 bytes/sec is due to the additional overheads involved with forwarding the packets through the Router and the various Graph nodes.

Somewhat unexpectedly, there appears to be larger differences at very large payload lengths along a fairly unpredictable curve. However, as these are still fairly small changes in total

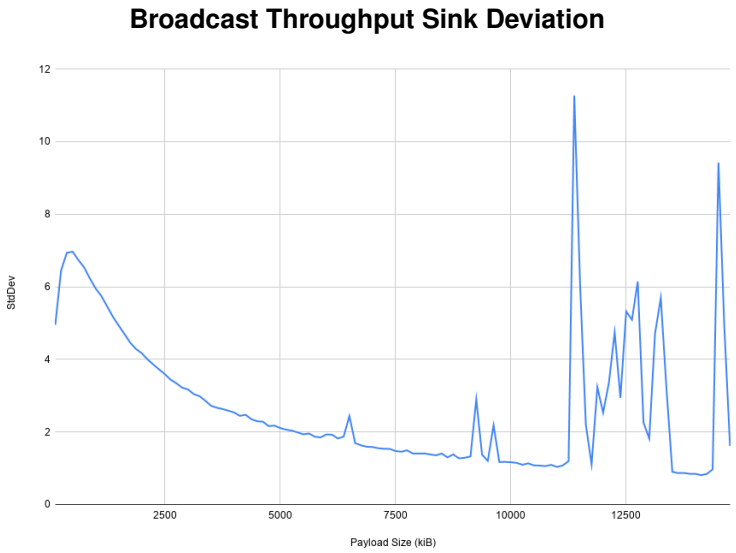


Figure 6.6: The throughput deviation between the Sink nodes and the Source. From data in Table B.3

throughput when compared to the source, this is unlikely to cause any issues during actual operation.

6.7.2 AnyCast

In anycast mode, the router randomly selects a forwarding node to transfer each message to, omitting all others (if there are any; broadcast and anycast are identical when only one forward address is known). Again, for each iteration, the router and nodes were shut down and reset, and the total payload length was increased by 1000 bytes each cycle.

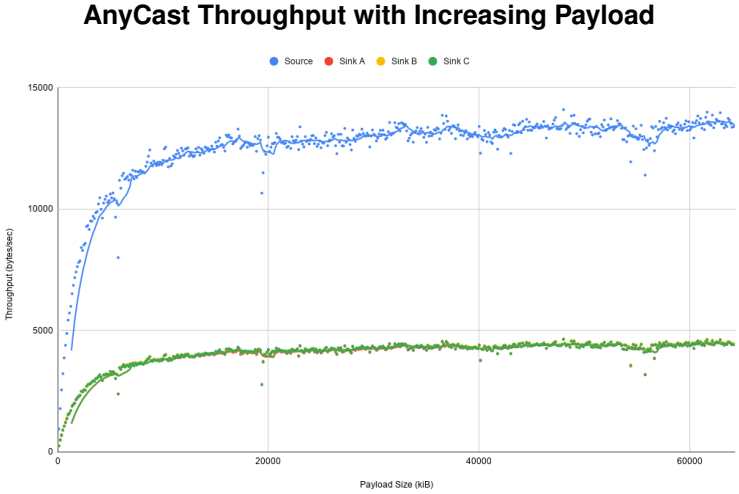


Figure 6.7: From data in Table B.5

Once again, transfer rates were monitored at both the source and sink nodes via the modified pv binary (See 6.4.2) then graphed against one another to produce Figure 6.7. Furthermore,

as the throughput sum of all sink nodes should be approximately equal to the throughput of the source node, we can calculate and determine a throughput difference to further characterise this mode; the results of which can be seen in Figure 6.8.

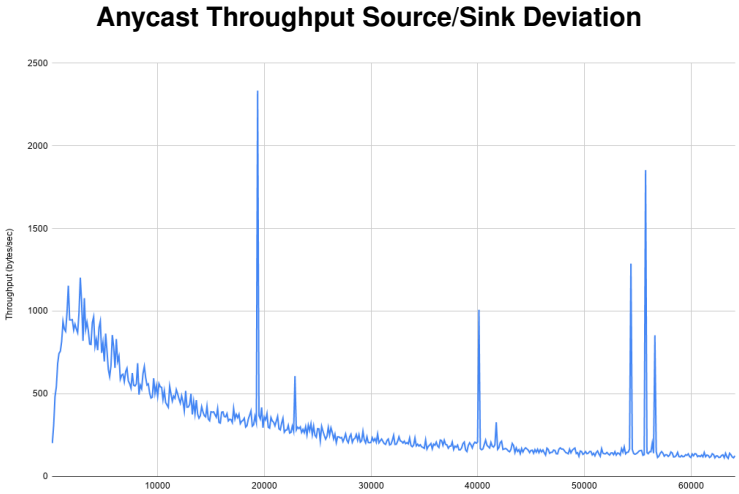


Figure 6.8: From data in Table B.5

The anycast selection in the Router is handled by the system `rand()` function, which is driven from the linux standard entropy source. As such, this provides a fairly uniform distribution of random values, such that the random selection presents an approximately even distribution of load over the sinks in this test.

It is expected that larger differences in individual throughput would be seen if extremely large numbers of sinks were used, but as the random source is designed to provide an even distribution, chasing this deviation was deemed to be overly complicated to achieve and has been omitted from these tests.

6.7.3 RoundRobin

In round-robin mode, the router attempts to forward each message to each forward table recipient *in turn*, looping back the start of the list each time around.

On its own, this presents a useful load-balancing mechanism for situations where the target throughput is not available at a particular point in the processing chain, and a number of nodes is instead ganged together to handle the workload. This is an extremely common pattern across the space of distributed and sequential processing, and can be seen readily in NLP literature via Apache HADOOP projects, as well as a wide range of other high-workload tasks, including web services for high performance, low data-per-transaction work.

In Figure 6.9 we see the three curves demonstrate the expected behaviour here where each sink receives approximately half of the throughput seen at the source. This is further evidenced in Figure 6.10 where the throughput discrepancy normalises around 130 bytes/second total (or

approximately 65 bytes/second loss per sink).

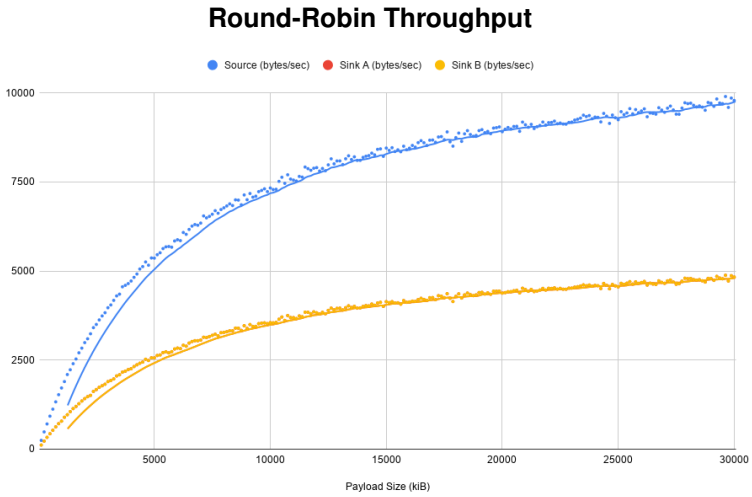


Figure 6.9: Round-robin routing options distribute the messages between all available sink nodes from the forward list. The data presented here demonstrates that the throughput on the sinks individually is precisely half the throughput seen at the source, less some measurement noise. From data in Table B.4

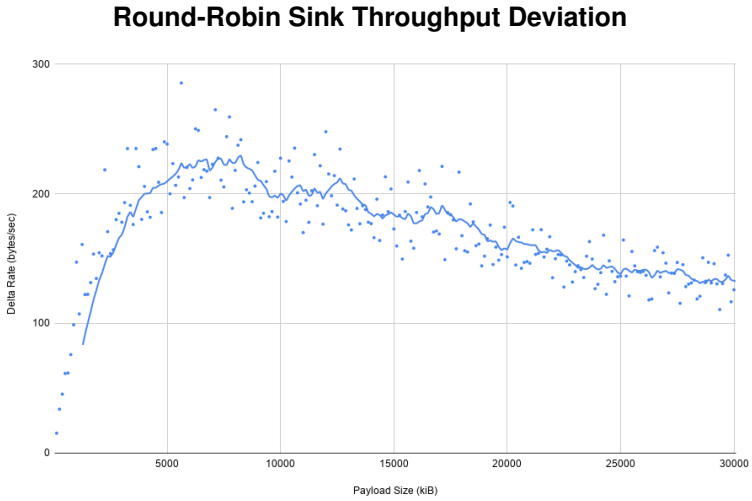


Figure 6.10: Taking the source throughput and subtracting the sub of all sink nodes gives a reasonable measure of how much performance is lost through the transport layer and router. In this case, after the measurement error at the lower end (sub-500,000 bytes) the loss normalises around 130 bytes/second loss in processing. From data in Table B.4

As the workload required to achieve round-robin operation is significantly lighter than that of the broadcast mode, we can see there is a correspondingly more smooth set of throughput curves.

Internally, the Router maintains a value in each context which points to the next offset in the forward list, such that transmission can be initiated *immediately* upon reception of a packet, and

once processed, the value is simply incremented ready for the next transmission. By contrast, broadcast modes require that the Router touches all forward addresses and their corresponding context structures to get each destination in turn for forwarding operations - a clearly much more heavyweight operation, causing the larger jitter presenting in Figure 6.6.

6.8 Runtime Operation

Now that the individual router modes have been characterised, the following sections will detail practical operations and configurations for use in actual data processing. The workloads in these examples were artificial so the end results bear no scientific worth by themselves, but serve to provide as close to a simulated task as possible.

OpenNLP was used to test the following configurations due to its highly modular nature; each process is encapsulated into a single command which accepts input and output streams or files. It is this high modularity that lends OpenNLP to GraphIPC operation, as workflows can be executed as discrete processes, and thus, be wrapped with the graph binary.

NLTK and GATE could have also been used for these tests, as both also follow a generally modular design, but during this testing, it was OpenNLP which provided sufficient pre-trained data to enable some of the subprocesses to function. Without having to first train a number of models or load particular databases, other frameworks would be more difficult to get up and running for the purposes required here.

Another option for these tests could have been to integrate with a larger package which encompasses multiple operations internally; effectively extending the package with GraphIPC aware interfaces.

SketchEngine somewhat straddles the line between a toolkit and a workflow system, where the workflow itself is implicit in the configurations available in the tool. As such, integration with such a tool would be much more involved, as the interfaces between operations are defined as part of the wider tool itself rather than requiring all input or output operations to be funnelled through system interfaces.

Furthermore, while the processes here have been aimed at scientific use cases in natural language processing, the underlying framework is intended to be more general, and moving the focus to the integration of a specific tool would change the direction of the work.

6.8.1 ‘Bus’ Operations

An extremely common pattern for NLP pipelines and linux terminal usage in general is to chain a series of processes together through their standard input and output pipes through the use of the ‘bar’ or ‘pipe’ operator in a terminal shell. This causes the standard output of one process

to be fed directly into the standard input of another through the use of an anonymous pipe as demonstrated here:

```
shell > processA | processB | processC
```

There is a clear basic parallel here to chaining a flat series of nodes together in GraphRouter, creating much the same sequence. However, as it is possible to send data from one node *back to itself*, some protection against stream confusion where messages endlessly loop the Graph wrapping binary will spawn a clone of the first process to handle messages from the new source address (See listing and graphic in Figure 6.11).

This kind of looped processing needs to be carefully managed by the user, however, as using this feature can quickly cause very large numbers of processes to be spawned, each with new addresses and route table entries.

It is not expected that this will be a particularly frequently used pattern for this design, due to the difficult nature of the configuration, but rather is here as a kind of safety net, whereby processes will allow multiple input connections without corrupting the output of the original process with other data.

To enable this, the node itself must also be in `--demux spawn` mode, otherwise the default `--demux noop` and `--demux merge` will pass the message through directly to the inner (optionally mangling the incoming address to match its own in the case of `merge`, effectively masquerading both flows as the same).

```
   Address Table:
|—> 00002000 {broadcast} to { 00002001 }  74.00 B 51.00 B Packets (2/3) BOUND
|—> 00001000 {broadcast} to { 00002000 }  51.00 B 0.00 B  Packets (3/0) BOUND
|—> 00002001 {broadcast} to { 00003000 }  0.00 B  74.00 B Packets (0/2) BOUND
```

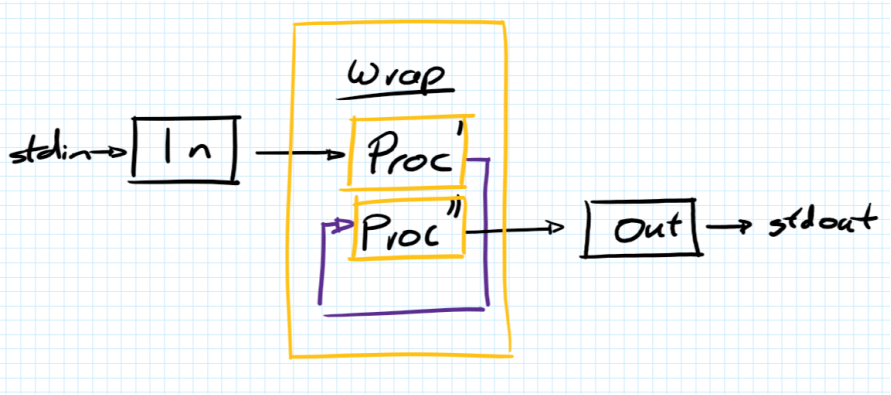


Figure 6.11: Two actual processes wrapped in the same node, one serving the base address $0x2000$, and the other providing $0x2001$ as a local address. Note that the route table includes a route to essentially the same node, but on a different address.

6.8.2 Flow Map and Reduce

To best illustrate how the functions described in the earlier sections of this chapter can be orchestrated and combined together to produce processing workflows the following two examples have been chosen; first, an example of a normal NLP workflow (See Figure 6.12), expressed as a graph all at once, rather than the usual separation of discrete processes. Then second, a real example of a workflow performed as part of a previous research effort (See Figure 6.14); although again, this was originally performed as a series of distinct steps, rather than the continuous flow presented here.

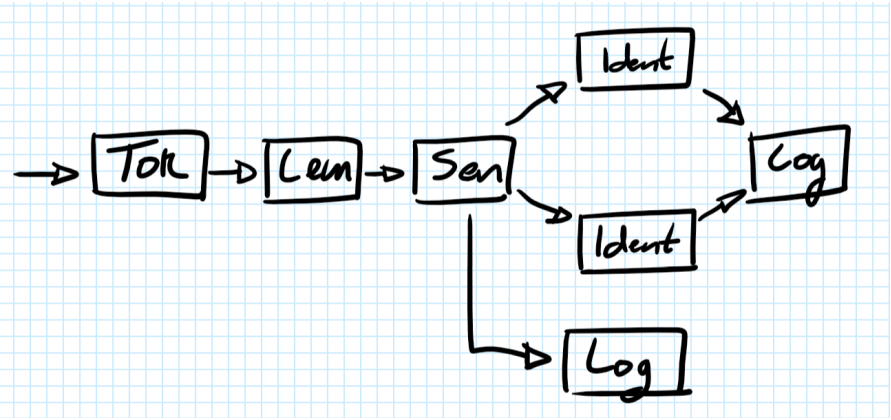


Figure 6.12: An example OpenNLP processing flow, using Tokenize, Lemmatize, Sentence Detection, Language Identification and Logging in ways common to natural language processing. As the possibilities for arranging the data flow here are effectively infinite, this particular arrangement has been chosen to demonstrate as many of the GraphIPC capabilties at once, rather than for being a practical real-world analysis.

While it should be fairly obvious how a number of identical streams can be produced from a single node by simply connecting multiple other nodes to it, there are in fact two ways to combine multiple streams and 'close the diamond' in the processing flow. The first is the simpler to understand, but requires more data to be present in the messages themselves to result in a useful output; the second, however results in simple message contents, but relies on the processes being aware or capable of using two input streams at once, adding complexity to the process.

The configuration shown in 6.13 demonstrates the latter, whereby a single node provides a data stream into the graph via its link to the standard input pipe from, and the comm binary is leveraged to recombine two independent streams through two nodes acting as 'shims' to join graph data to the standard linux/unix pipe mechanisms.

In their papers [2, 4, 32, 42], Baron, Rayson and Archer et.al explore language identification proceses through before *and* after a word normalisation phase, and comparing the two outputs, it is possible to get much more correct results from historical corpora where word spellings have

```
Address Table:
  |—> 00001000 {broadcast} to { 00002000 00003000 }   381.00 B    0.00 B      Packets (22/0)  BOUND
  |—> 00002000 {broadcast} to { 00004000 }            424.00 B    381.00 B    Packets (20/22) BOUND
  |—> 00005000 {drop}      to { none }                0.00 B      424.00 B    Packets (0/20)  BOUND
  |—> 00003000 {broadcast} to { 00005000 }            424.00 B    381.00 B    Packets (20/22) BOUND
  |—> 00004000 {drop}      to { none }                0.00 B      424.00 B    Packets (0/20)  BOUND
```

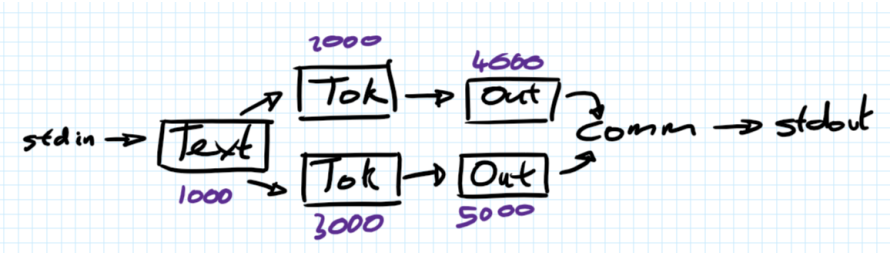


Figure 6.13: An example of the address table in the GraphRouter for a diamond relationship between nodes. Note that this particular example was running the comm binary between the outputs of nodes 4000 and 5000 to compare the two outputs rather than using a dedicated graph-aware binary. Nodes 2000 and 3000 were running two different tokenenisation models for comm to compare. The diagram below the configuration also illustrates this mapping.

changed over time, while also still removing mis-spellings and typographical errors, culminating in the VARD2[3] tool for spelling variation.

In Figure 6.14, the graph has been configured in such a way that this process may be fully realised as a parallel process, with a single word token source spanning both chain configurations, followed by a comparison stage (Note that in this example, the checking stage is simulated, but could be created through judicious use of the Linux/UNIX comm binary[45] to compare the two streams).

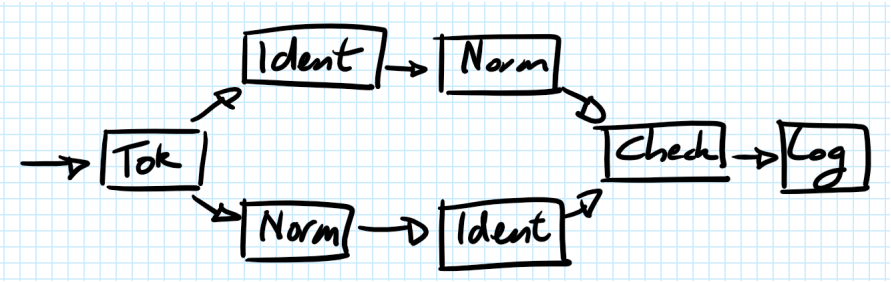


Figure 6.14: This GraphIPC configuration demonstrates one option for processing historical corpora in such as way as to correctly identify the language or language-type despite the presence of mis-spelling and transcription errors.

Configuring the router manually in the manner that has been described here give the user maximum freedom to describe any sequence of operations; but it does require considerable pre-planning. Each link must be described and each operating mode, both at the Router and at the nodes must be defined before processing the data stream.

As has been previously discussed as part of Chapter 4 workflow languages have been

available for a while, and are capable of simulating the kind of flows demonstrated here. This is normally achieved through the use of intermediate input/output files and a flow unwrapping system where parallel operations are reduced to sequential operations.

Systems such as SnakeMake[46] which have gained popularity in other scientific fields through its extremely generalised nature, makes these intermediate files very evident, including them explicitly as part of the build sequences, and allowing users to take advantage of this effectively side-effect-free logging.

6.9 Limitations and Extensions

As the implementation presented here is a prototype, certain notable features have been omitted which would be desirable as part of future, full implementations.

With the increase in systems using control groups (cgroups) mechanisms to isolate processes from one another - as used extensively in Docker, for example - being able to execute multiple routers on the same host in some coordinated manner is desirable.

It is hoped that by implementing this as a kernel module - as would be the case for a complete implementation - it would be simple to have a single, connected router instance underlying all containers on the host, but there may well be cases where the graph itself *also* needs to be contained, and having an extensions to the scheme along the lines of what was described in Section 5.2.1 would provide a mechanism for this.

6.9.1 Build System Integration

With a stable base to build upon, the current prototype could be incorporated into a build tool - such as CWL or SnakeMake - which through their existing mechanisms for describing parallel flows could easily be made to produce actually parallel graphs rather than the apparent parallelism they often employ.

All the tests in this chapter were orchestrated through regular Bash scripts, however (these can be viewed in the code repository, should they be useful), so creating and configuring networks does not require any of the workflow languages or tools to actually work.

6.9.2 Transport Layer

The prototype implementation for the router portion of this design is presented here as built upon loopback TCP sockets. In a system implementation of this, it would be expected to be a full kernel module, and as such would need no additional transport layer, as with direct access to individual processes memory, the routing module would either be able to shuttle data directly into the memory of the receiver, or as an alternative, would be able to utilise a netlink socket

class to ensure that packet framing is preserved over the kernel/user boundary, presenting a packet interface which any socket-capable process would the be able to tap into.

6.9.3 Multiple Host or Nested Hosting

While this prototype would quite happily execute within a process container or sandbox environment, any linkage to the rest of the host outside the container would have to be conducted through an external link, losing the addressing information from the container router. This problem is also equally present in cases where multiple hosts are to be connected together. Without a robust addressing scheme to span multiple hosts, it would be quite easy for multiple nodes to have identical addresses, causing problems when any needs to transmit data from one router to another.

Furthermore, even if the addressing problem could be resolved, there would exist a forward table problem between hosts, as there is currently no mechanism to identify a route to send a message out from if the destination is not the intended recipient (as per traditional packet forwarding networks). This would have to be addressed by an extension to the system to incorporate these situations.

However, these problems, while considerable, are not completely unknown to the networking literature, and have been largely addressed by prior work in areas such as general routing protocols (BGP, MPLS, and others) in addition to addressing systems (DHCP, DNS) so future work could integrate this into the graph to aid these network spanning problems.

Tunnelling one router to another, for example, may be a solution to the addressing issue, or providing specific address ranges for specific routers, as seen by the original IPv4 specification, where given high-range prefixes are intended to be used for specific purposes on the network.

6.10 Summary

The testing done in this chapter has characterised the performance and abilities of the GraphIPC prototype, and has attempted to address any problems with the implementation at the time of writing.

Additionally, it has been shown in the previous chapter (See Section 5.3.3), then further evidenced here that the internal mechanics of the GraphIPC router and wrapping process are performant; namely in the areas of address lookup and forward destination resolution.

The various policies for message forwarding have been described, illustrated and tested in isolated configurations, as well as a more complete application focused approach exploring the possibilities of the GraphIPC network as a whole as a tool; this included the practicalities of specifying the graph connections, as well as the limitations therein.

Footnotes

[41] See https://www.archlinux.org/ for details

[42] See https://ubuntu.com/ for details

[43] https://travis-ci.com/ - performs automated code builds in containers to validate version controlled software, usually from GitHub repositories.

[44] This is currently (at the time of writing) hosted on GitHub at https://github.com/JohnVidler/GraphIPC, see this URL for the latest sources, scripts and tools. If this project somehow outlasts GitHub, sources may also be obtained at https://johnvidler.co.uk/.

[45] See https://linux.die.net/man/1/comm for details, although the suggested syntax would be "comm -23 <(Graph -a 1000 -o) <(Graph -a 2000 -o) | Graph -a 3000 -i" such that nodes 1000 and 2000 can be used to funnel data into comm, then out via 3000

[46] See https://snakemake.readthedocs.io/en/stable/ for the SnakeMake documentation

Chapter 7

Conclusion

In the latter portion of the opening chapter of this thesis, four research questions were stated. In Chapter 3 *What would a modern workload-focused approach look like for local message passing?* (RQ1) was explored, with a focus on how processes in modern workloads actually interact. This included work of a more abstract nature to identify the shape of the rest of the work (Sections 3.2.1 and 3.3.1), concluding with a discussion around Multi-Path IPC in the general case (Section 3.3.3). A further exploration of the dominant design patterns for high-speed computing; Map and Reduce (Section 3.4) along with a case study involving local GPU computing (Section 3.5) exploring the use and characteristics of the hardware all provided further insights into how modern workloads are handled.

What affordances are there to use the modern heterogeneous systems more effectively for analysis tasks? (RQ2) was addressed with a case study subsequently published as "Dealing With Big Data Outside Of The Cloud: GPU Accelerated Sort" [54] and discussed as part of Section 3.5.

In this section and paper, I demonstrate that using even traditionally sub-optimal approaches can give significant processing performance gains when used at large scales (See Figures 3.16 and 3.17). The 'swap sort' algorithm used to generate the sorted lists in the paper is extremely inefficient on normal computing hardware, and would be inappropriate for use without the vast multiplication of effort that running on a GPU can generate.

What workflow elements are required to support large scale data and text analytical tasks? (RQ3) has been addressed as part of the design process of building GraphIPC, as detailed in Chapter 4, with the introduction and identification of the components required to build GraphIPC; namely, the Mux (Section 4.5.4), Demux (Section 4.5.5), and Bus (Section 4.5.1) message handling components, along with the more familiar Map and Reduce design patterns (Sections 4.5.2 and 4.5.3, respectively) from high-performance computing.

What of systems design attempts from the last 30 years is actually applicable or appropriate

for modern workloads? (RQ4) was explored in the Related Work (Chapter 2) and Analysis (Chapter 3). In these chapters, I examined a number of different kernel designs with a focus on how they interact with their various components, and how this affects the overall design of the system. In the Tesselation design (Section 2.12.1) in particular, the focus on the general distribution of processing load affected the rest of this work; this along with the Barrelfish (Section 2.12.4) design set the scope for the design of GraphIPC's routing layer.

7.1 Novelty

The model and prototype of GraphIPC presented here describes a novel way for processes to interact. It describes a way for messages to be transferred between 'nodes' asynchronously, but in-order, and along both unicast and multicast paths.

The two binaries `GraphRouter` and `Graph` encompass two halves of the model, with the Router shuttling data around the system to instances of Graph, and the Graph presenting a bridging component between existing software and provides a demonstrator for working with other existing tools.

This brings many traditionally networking concepts down into the interprocess communication space, enabling more complete workflows to be realised in increasingly parallel, heterogeneous environments using existing tooling.

Specific extensions to existing NLP workflows have been demonstrated in Section 6.8.2, where the addition of GraphIPC to the processing sequence would introduce new possibilities for analysis, including real-time analysis.

The workflows presented here using the prototype have been given a natural language processing focus, but the model is designed with general purpose processing in mind, and as such should be applicable to any complex workflow interactions.

In contrast to existing systems, the model describes continuous, live interaction between processes, enabling both injection and tapping of messages while the system is operating without affecting the communication in progress.

7.2 Utility

The prototype presented here needs to, at minimum, perform the same functions as well as the existing system interfaces to be 'useful' in a real sense; ideally, this would include improvements to the overall performance or features available as well.

Examining the throughput of each subcomponent of the GraphIPC prototype (Design details in Section 4.5, evaluated in Section 6.7 onwards) shows some drop in overall performance of

the individual links. This, however, is to be expected, as the additional routing task between each message is required and takes some processing time to achieve.

Furthermore, as the prototype presented here exists purely in user space, and takes no advantage of kernel space structures or tricks; it is therefore using the kernel 'blind', with the transport layer being unaware of what the prototype is actually doing with the data and purely shuttling it in a TCP loopback socket between processes and the router.

With the router process in userspace, each packet must make a minimum of 4 kernel boundary crossings which will significantly reduce the throughput. This particular limitation was identified and discussed in Section 5.5 and illustrated in Figure 5.17.

However, with these known issues GraphIPC remains a practical tool for routing messages around an NLP workflow, as the actual traffic seen on this kind of network will tend to have low bandwidth requirements beyond the first stages of processing, as an unfortunate consequence of the difficulties in processing natural languages is fairly long run times for analysis, even when split among several stages.

An interesting point to note at this stage however, is that the pipe-to-socket test described in Section 6.6 - a close approximation to what GraphIPC is actually doing, but hidden by the Graph binary wrapper - achieves performance which plateaus around 15kB/s while performing a one-to-one link, whereas GraphIPC still reaches 13.5kB/s while performing anycast communication. Note that internally, anycast communication is the most lightweight form of routing for GraphRouter, as it only has to grab the next target process based on a random offset in the target table and continue, whereas all other forms (Broadcast and Round-Robin) require pointer updates to track the current policy state.

This then, is only a modest reduction in performance, while offering all the features the GraphIPC Router describes to provide asynchronous graph-form messaging. Therefore, I assert that this prototype - although slightly suboptimal - remains a useful tool to the NLP community.

7.3 Significant Contributions

This work has opened the opportunity for other work - both my own and other researchers - for exploring novel ways to interact with data streams between process stages, predominantly in natural language processing pipelines, but also in other areas of data processing.

So far, the prototype has only been applied to working with 'bare' OpenNLP tooling, and with workloads which are merely representative of normal workloads for natural language processing, rather than actual real-world tool chains[47].

Without the support of the features this prototype presents, dynamic connections between

processes in graph forms would not be possible on the systems available to developers and researchers today.

Furthermore, the inclusion of a tool (GraphWrap) to encapsulate legacy applications as part of the larger graph network allows for back-compatibility to tool chains, significantly augmenting their capabilities without requiring any additional work on the part of the tool authors.

7.4 Limitations and Further Work

The prototype presented in this thesis is just that - a prototype - further work would be required to take this to production quality, most notably as has been pointed out throughout; the Router component of GraphIPC would greatly benefit from becoming a kernel module.

7.4.1 Building as a Kernel Module

As a module, and thus part of the kernel, the router would have no need for the buffering support that the current prototype uses, as it would be able to directly shuttle data between the individual nodes using 'proper' netlink sockets.

The advantages this would present are twofold; first, as the current implementation is built entirely in userspace, each message delivery operation require *at least two* kernel boundary crossings. One to send each message to the router, and another one each to forward this on to the destination node or nodes.

The second advantage would be the access to packet framing via the netlink connection, as netlink preserves packet boundaries through its API, rather than the TCP link chosen here which drops the edges of frames requiring the applications themselves to track them manually adding significant overheads at both ends of the connections.

7.4.2 Networking and Protocol

Additionally, while the network protocol itself is quite simple, handling the parsing and generation of packets manually is generally not considered good practice anymore unless its needed for a particular reason. Instead a framework such as Google's Protocol Buffers[48], Cap'n Proto[49] or FlatBuffers[50].

All of these frameworks offer ways to describe messages external to the code, and have the serialisation and deserialisation routines for these formats to be generated as a pre-compile step. The main advantages for this is consistency and validation, as both sides are (nearly) guaranteed to be free of human error.

At the time of writing, FlatBuffers offers the best feature set to describe the messages GraphIPC uses, although this would have to be examined when any extension is undertaken.

7.4.3 Integration With a Zero-Copy Framework

Rather than relying on a custom routing process embedded in the kernel, there may be gains associated with moving this prototype to a component as part of a larger zero-copy framework, such as DPDK[29] or Netmap[46], where most of the work doing the heavy lifting for packet transfer has already been done, and can be shown to be extremely close to line-rate, saturating the links.

This would allow a GraphIPC subsystem to be embedded into a general-purpose system much more readily, whereas as it stands, the prototype must be run independantly of any other subsystem.

7.4.4 Remote Host and Nested Router Support

Related to the networking extensions, the router implementation currently makes a number of assumptions which may not be correct in all cases.

One such assumption is that there *is only one router* on each host. While in most cases this may well be true, certainly on general purpose machines (such as laptops, and desktops), in the case where this were to be deployed on a server, it is extremely likely that multiple routers would be desirable.

Given the continued rise of containers (and systems therein, such as Docker) and related process isolation techniques, it may be desirable to run a router within each container context, rendering each set of node addresses within a single control group space. This would enable local routers per container, at the expense of slightly more storage use for each container as tracking the additional address spaces would be required.

In situations like the one described here, it would then follow that it would likely be desirable to connect multiple router instances together directly, without an intermediate node proxying messages between them. This link should allow the address spaces of each to be mapped in some way to the other, thus allowing the 'remote' processes to send messages over the link. Equally, in situations such as clusters or web-based systems, it may also be desirable to run multiple independent routers and connect across entire hosts, to build a coherent system across a larger collection of hardware.

7.4.5 Workflow Tool Integration

Each of the additions here described adds some level of complexity to the overall design (with the possible exception of the kernel module integration), and as the complexity of controls at each point in the network increases, the need for some method for defining flows becomes more pressing.

Technologies such as Snakemake, CWL and others are readily available to work with GraphIPC, but have no mechanism to declare the routes required at present. Therefore, a particularly useful addition to this work would be to extend one of the existing workflow description tools (or workflow description languages) to use GraphIPC directly, ideally with sensible default configurations baked in to the systems to protect users from accidental misconfiguration.

Footnotes

[47]'Tool chains' are a bit of a misnomer here, as the 'chain' implies sequentiality, rather than for example, 'Tool Graphs' perhaps when referring to this style of processing in general? Such a term would be applicable to othertool kits which allow pseudo-graph flows too.

[48]https://developers.google.com/protocol-buffers

[49]https://capnproto.org/

[50]https://google.github.io/flatbuffers/

Chapter 8

Bibliography

[1] Laurence Anthony. *Laurence Anthony's Software*. http://www.laurenceanthony.net/software.html. [Online; retrieved 28th October 2017] (cit. on p. 48).

[2] Dawn Archer et al. "Guidelines for normalising early modern English corpora: decisions and justifications". English. In: *ICAME Journal* 39.1 (Mar. 2015). 2015. This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 3.0 License. (CC BY-NC-ND 3.0), pp. 5–24. ISSN: 1502-5462. DOI: 10.1515/icame-2015-0001 (cit. on p. 102).

[3] Alistair Baron and Paul Rayson. "VARD2: A tool for dealing with spelling variation in historical corpora". In: *Postgraduate conference in corpus linguistics*. 2008 (cit. on p. 103).

[4] Alistair Baron, Paul Rayson and Dawn Archer. "Word frequency and key word statistics in corpus linguistics". English. In: *Anglistik* 20.1 (2009), pp. 41–67. ISSN: 0947-0034 (cit. on p. 102).

[5] Andrew Baumann et al. "The multikernel: a new OS architecture for scalable multicore systems". In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. SOSP '09. Big Sky, Montana, USA: ACM, 2009, pp. 29–44. ISBN: 978-1-60558-752-3. DOI: http://doi.acm.org/10.1145/1629575.1629579. URL: http://doi.acm.org/10.1145/1629575.1629579 (cit. on pp. 23, 25).

[6] Edward Loper Bird Steven and Ewan Klein. *Natural Language Processing with Python*. O'Reilly Media Inc, 2009 (cit. on p. 9).

[7] Martin Bor, John Edward Vidler and Utz Roedig. "LoRa for the Internet of Things". In: 16 (2016), pp. 361–366. URL: https://www.researchgate.net/profile/John_Vidler2/publication/297731094_LoRa_for_the_Internet_of_Things/links/56e1893e08ae4bb9771ba9e3/LoRa-for-the-Internet-of-Things.pdf (cit. on p. ix).

[8] Silas Boyd-Wickizer et al. “Corey: an operating system for many cores”. In: *Proceedings of the 8th USENIX conference on Operating systems design and implementation*. OSDI’08. Berkeley, CA, USA: USENIX Association, 2008, pp. 43–57. URL: http://dl.acm.org/citation.cfm?id=1855745http://portal.acm.org/citation.cfm?id=1855741.1855745 (cit. on p. 26).

[9] *BYU corpora: billions of words of data: free online access*. https://corpus.byu.edu/. [Online; retrieved 28th October 2017] (cit. on p. 48).

[10] Daniel Cederman and Philippas Tsigas. “GPU-Quicksort: A practical Quicksort algorithm for graphics processors”. In: *J. Exp. Algorithmics* 14 (Jan. 2010), 4:1.4–4:1.24. ISSN: 1084-6654. DOI: 10.1145/1498698.1564500. URL: http://doi.acm.org/10.1145/1498698.1564500 (cit. on p. 49).

[11] Rishav Chakravarti et al. *CFO: A Framework for Building Production NLP Systems*. 2019. arXiv: 1908.06121 [cs.CL] (cit. on p. 10).

[12] Hamish Cunningham. “GATE, a General Architecture for Text Engineering”. In: *Computers and the Humanities* 36.2 (2002), pp. 223–254. ISSN: 1572-8412. DOI: 10.1023/A:1014348124664. URL: https://doi.org/10.1023/A:1014348124664 (cit. on pp. 7, 8).

[13] Yangdong Steve Deng. “IP routing processing with graphic processors”. In: *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)* (Mar. 2010), pp. 93–98. DOI: 10.1109/DATE.2010.5457229. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5457229 (cit. on pp. 49, 54).

[14] *Docker - Build, Ship, and Run Any App, Anywhere*. https://www.docker.com/. [Online; retrieved 28th October 2017] (cit. on p. 3).

[15] *Documentation — MQTT*. http://mqtt.org/documentation. [Online; retrieved 6th August 2018] (cit. on p. 11).

[16] ECMA. *ECMA-404 - The JSON Data Interchange Format*. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf. 2013 (cit. on p. 62).

[17] Dawson R Engler, M Frans Kaashoek and J. O’Toole. *Exokernel*. New York, New York, USA: ACM Press, 1995, pp. 251–266. ISBN: 0897917154. DOI: 10.1145/224056.224076. URL: http://portal.acm.org/citation.cfm?doid=224056.224076 (cit. on p. 22).

[18] Clark C. Evans. *YAML - YAML Ain’t Markup Language*. http://yaml.org/. 2011 (cit. on p. 62).

[19] Sylvain Frey et al. “It Bends but Would it Break? Topological Analysis of BGP Infrastructures in Europe”. In: *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2016, pp. 423–438. URL: https://eprints.soton.ac.uk/412811/1/EuroSnP.pdf (cit. on p. ix).

[20] Sadayuki Furuhashi. *MessagePack - It's like JSON, but fast and small*. http://msgpack.org. 2013 (cit. on p. 62).

[21] *GATE.ac.uk*. https://gate.ac.uk/. [Online; retrieved 28th October 2017] (cit. on p. 48).

[22] Adam Greenhalgh, Mark Handley and Felipe Huici. "Flowstream Architectures 1 Introduction 2 Flowstream Architectures". In: *Computer* 17.2009 (), pp. 1–5 (cit. on p. 2).

[23] Adam Greenhalgh et al. "Flowstream Architectures." In: *ECEASST* (2009), pp. –1–1 (cit. on p. 2).

[24] BSON Group. *BSON - Binary JSON*. http://bsonspec.org/. 2015 (cit. on p. 62).

[25] John Hardy et al. "Shapeclip: towards rapid prototyping with shape-changing displays for designers". In: *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. ACM. 2015, pp. 19–28 (cit. on p. ix).

[26] Nancy Ide, Keith Suderman and Jin-Dong Kim. "Mining Biomedical Publications With The LAPPS Grid". In: *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*. Miyazaki, Japan: European Language Resources Association (ELRA), Dec. 7. ISBN: 979-10-95546-00-9 (cit. on p. 10).

[27] Nancy Ide et al. "The Language Applications Grid". In: *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14)*. Ed. by Nicoletta Calzolari (Conference Chair) et al. Reykjavik, Iceland: European Language Resources Association (ELRA), May 2014. ISBN: 978-2-9517408-8-4 (cit. on p. 10).

[28] University of Southern California Information Sciences Institute. *Transmission Control Protocol*. https://tools.ietf.org/html/rfc793. 1981 (cit. on p. 18).

[29] Intel. *Data Plane Development Kit*. https://www.dpdk.org/. [Online; retrieved 18th February 2020] (cit. on pp. 15, 111).

[30] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *arXiv preprint arXiv:1801.01203* (2018). eprint: 1801.01203. URL: https://spectreattack.com/spectre.pdf (cit. on p. 54).

[31] Johannes Köster and Sven Rahmann. "Snakemake—a scalable bioinformatics workflow engine". In: *Bioinformatics* 28.19 (Aug. 2012), pp. 2520–2522. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/bts480. eprint: http://oup.prod.sis.lan/bioinformatics/article-pdf/28/19/2520/819790/bts480.pdf. URL: https://doi.org/10.1093/bioinformatics/bts480 (cit. on p. 9).

[32] Anu Lehto et al. "Improving the precision of corpus methods: The standardized version of Early Modern English Medical Texts". English. In: *Early Modern English Medical Texts*. Ed. by Irma Taavitsainen and Päivi Pahta. John Benjamins, 2010, pp. 279–289. ISBN: 978 90 272 1177 4 (cit. on p. 102).

[33] *liblfds.org*. https://liblfds.org/. [Online; retrieved 23rd October 2017] (cit. on p. 45).

[34] Moritz Lipp et al. “Meltdown”. In: *arXiv preprint arXiv:1801.01207* (2018). eprint: 1801.01207. URL: https://meltdownattack.com/meltdown.pdf (cit. on p. 54).

[35] Jed Liu et al. “Fabric: a platform for secure distributed computation and storage”. In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. SOSP ’09. Big Sky, Montana, USA: ACM, 2009, pp. 321–334. ISBN: 978-1-60558-752-3. DOI: http://doi.acm.org/10.1145/1629575.1629606. URL: http://doi.acm.org/10.1145/1629575.1629606 (cit. on pp. 27, 28).

[36] Rose Liu et al. “Tessellation: Space-Time Partitioning in a Manycore Client OS”. In: () (cit. on p. 20).

[37] Carlos Aguilar Melchor et al. “High-Speed Private Information Retrieval Computation on GPU”. In: *Proceedings of the 2008 Second International Conference on Emerging Security Information, Systems and Technologies*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 263–272. ISBN: 978-0-7695-3329-2. DOI: 10.1109/SECURWARE.2008.55. URL: http://portal.acm.org/citation.cfm?id=1447563.1447928 (cit. on pp. 49, 54).

[38] a. Mirtchovski, R. Simmonds and R. Minnich. “Plan 9 - an integrated approach to grid computing”. In: *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* 00.C (2004), pp. 273–280. DOI: 10.1109/IPDPS.2004.1303349. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1303349 (cit. on p. 31).

[39] Edmund B. Nightingale et al. “Helios: heterogeneous multiprocessing with satellite kernels”. In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. SOSP ’09. Big Sky, Montana, USA: ACM, 2009, pp. 221–234. ISBN: 978-1-60558-752-3. DOI: http://doi.acm.org/10.1145/1629575.1629597. URL: http://doi.acm.org/10.1145/1629575.1629597 (cit. on pp. 29, 30).

[40] Ruslan Nikolaev and Godmar Back. “VirtuOS: An Operating System with Kernel Virtualization”. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 2013. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522719 (cit. on p. 3).

[41] Nebojša Tijanić (editors) Brad Chapman John Chilton Michael Heuer Andrey Kartashov Dan Leehr Hervé Ménager Maya Nedeljkovich Matt Scales Stian Soiland-Reyes Luka Stojanovic Peter Amstutz Michael R. Crusoe. “Common Workflow Language, v1.0. Specification, Common Workflow Language working group.” In: (2016). DOI: https://doi.org/10.6084/m9.figshare.3115156.v2. URL: https://w3id.org/cwl/v1.0/ (cit. on p. 10).

[42] Scott Songlin Piao et al. “A time-sensitive historical thesaurus-based semantic tagger for deep semantic annotation”. English. In: *Computer Speech and Language* 46 (Nov. 2017), pp. 113–135. ISSN: 0885-2308. DOI: 10.1016/j.csl.2017.04.010 (cit. on p. 102).

[43] Version Wire Protocol. “OpenFlow Switch Specification”. In: *Wire* 0 (2009), pp. 1–42 (cit. on p. 2).

[44] Layali Rashid, WessamM. Hassanein and MoustafaA. Hammad. “Analyzing and enhancing the parallel sort operation on multithreaded architectures”. English. In: *The Journal of Supercomputing* 53.2 (2010), pp. 293–312. ISSN: 0920-8542. DOI: 10.1007/s11227-009-0294-5. URL: http://dx.doi.org/10.1007/s11227-009-0294-5 (cit. on p. 49).

[45] Paul Rayson. *CQPweb Main Page*. https://cqpweb.lancs.ac.uk/. [Online; retrieved 28th October 2017] (cit. on p. 48).

[46] Luigi Rizzo. *Netmap*. https://github.com/luigirizzo/netmap. [Online; retrieved 18th February 2020] (cit. on pp. 15, 111).

[47] Jonathan S. Shapiro, Jonathan M. Smith and David J. Farber. “EROS: A Fast Capability System”. In: *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*. SOSP ’99. Charleston, South Carolina, USA: Association for Computing Machinery, 1999, 170–185. ISBN: 1581131402. DOI: 10.1145/319151.319163. URL: https://doi.org/10.1145/319151.319163 (cit. on p. 4).

[48] *Sketch Engine — language corpus management and query system*. https://www.sketchengine.co.uk/. [Online; retrieved 28th October 2017] (cit. on p. 48).

[49] Weibin Sun, Robert Ricci and Matthew L. Curry. “GPUstore”. In: *Proceedings of the 5th Annual International Systems and Storage Conference on - SYSTOR ’12*. New York, New York, USA: ACM Press, 2012, pp. 1–12. ISBN: 9781450314480. DOI: 10.1145/2367589.2367595. URL: http://dl.acm.org/citation.cfm?id=2367595 (cit. on pp. 49, 54).

[50] Weibin Sun, Robert Ricci and Matthew L Curry. “GPUstore: harnessing GPU computing for storage systems in the OS kernel”. In: *Proceedings of the 5th Annual International Systems and Storage Conference*. SYSTOR ’12. New York, NY, USA: ACM, 2012, 6:1–6:12. ISBN: 978-1-4503-1448-0. DOI: 10.1145/2367589.2367595. URL: http://doi.acm.org/10.1145/2367589.2367595 (cit. on p. 3).

[51] Chandramohan A Thekkath et al. “Implementing network protocols at user level”. In: *SIGCOMM Comput. Commun. Rev.* 23.4 (Oct. 1993), pp. 64–73. ISSN: 0146-4833. DOI: 10.1145/167954.166244. URL: http://doi.acm.org/10.1145/167954.166244 (cit. on p. 3).

[52] Pedro Trancoso, Despo Othonos and Artemakis Artemiou. “Data parallel acceleration of decision support queries using Cell/BE and GPUs”. In: *Proceedings of the 6th ACM conference on Computing frontiers - CF ’09* (2009), p. 117. DOI: 10.1145/1531743.1531763. URL: http://portal.acm.org/citation.cfm?doid=1531743.1531763 (cit. on p. 3).

[53] John Vidler and Stephen Wattam. “Keeping Properties with the Data CL-MetaHeaders-An Open Specification”. In: *CMLC-5+BigNLP* (2017). URL: https://ids-pub.bsz-bw.de/frontdoor/index/index/docId/6243 (cit. on pp. iii, viii, 10).

[54] John Vidler et al. “Dealing With Big Data Outside Of The Cloud: GPU Accelerated Sort”. In: *Challenges in the Management of Large Corpora. CMLC-2*. Reykjavik, 2014, p. 21. URL: http://www.lrec-conf.org/proceedings/lrec2014/workshops/LREC2014Workshop-CMLC2Proceedings-rev2.pdf (cit. on pp. iii, viii, 107).

[55] W3C. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. http://www.w3.org/TR/REC-xml/. 2008 (cit. on p. 62).

[56] I Watson and J Gurd. “A Practical Data Flow Computer”. In: *Computer* 15.2 (Feb. 1982), pp. 51–57. ISSN: 0018-9162. DOI: 10.1109/MC.1982.1653941 (cit. on p. 2).

[57] Stephen Wattam et al. “Experiences with Parallelisation of an Existing NLP Pipeline : Tagging Hansard”. In: *Proceedings of The 9th edition of the Language Resources and Evaluation Conference*. 2014 (cit. on p. 49).

[58] *Welcome to Apache Hadoop!* https://hadoop.apache.org/. [Online; retrieved 22nd October 2017] (cit. on pp. 11, 45).

[59] *Wmatrix corpus analysis and comparison tool*. http://ucrel.lancs.ac.uk/wmatrix/. [Online; retrieved 28th October 2017] (cit. on p. 48).

[60] *WordSmith Tools home page*. http://www.lexically.net/wordsmith/. [Online; retrieved 28th October 2017] (cit. on p. 48).

Appendix A

List of Figures

Appendix B

Data Tables

B.1 Hardware Specification of Test Environment

```
$ lscpu
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   39 bits physical, 48 bits virtual
CPU(s):                          8
On-line CPU(s) list:             0-7
Thread(s) per core:              2
Core(s) per socket:              4
Socket(s):                       1
NUMA node(s):                    1
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           142
Model name:                      Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
Stepping:                        10
CPU MHz:                         1721.310
CPU max MHz:                     3400.0000
CPU min MHz:                     400.0000
BogoMIPS:                        3601.00
Virtualization:                  VT-x
L1d cache:                       128 KiB
L1i cache:                       128 KiB
L2 cache:                        1 MiB
L3 cache:                        6 MiB
NUMA node0 CPU(s):               0-7
Vulnerability L1tf:              Mitigation; PTE Inversion; VMX conditional cache flushes, SMT vulnerable
Vulnerability Mds:               Vulnerable: Clear CPU buffers attempted, no microcode; SMT vulnerable
Vulnerability Meltdown:          Mitigation; PTI
Vulnerability Spec store bypass: Mitigation; Speculative Store Bypass disabled via prctl and seccomp
Vulnerability Spectre v1:        Mitigation; usercopy/swapgs barriers and __user pointer sanitization
Vulnerability Spectre v2:        Mitigation; Full generic retpoline, IBPB conditional, IBRS_FW, STIBP
     conditional, RSB filling
Flags:                           fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36
     clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art
     arch_perfmon pebs bts rep_good nopl xto
                                 pology nonstop_tsc cpuid aperfmperf tsc_known_freq pni pclmulqdq dtes64
     monitor ds_cpl vmx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt
     tsc_deadline_timer aes xsave avx f16c
                                  rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb invpcid_single pti ssbd
     ibrs ibpb stibp tpr_shadow vnmi flexpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1 avx2 smep bmi2
     erms invpcid mpx rdseed adx sm
                                 ap clflushopt intel_pt xsaveopt xsavec xgetbv1 xsaves dtherm ida arat pln
     pts hwp hwp_notify hwp_act_window hwp_epp flush_l1d

$ callisto
    description: Notebook
    product: 20L7S00R00 (LENOVO_MT_20L7_BU_Think_FM_ThinkPad T480s)
    vendor: LENOVO
    version: ThinkPad T480s
    serial: PC0XDVFK
    width: 4294967295 bits
    capabilities: smbios-3.0 dmi-3.0 smp vsyscall32
    configuration: administrator_password=disabled chassis=notebook family=ThinkPad T480s power-on_password=
     disabled sku=LENOVO_MT_20L7_BU_Think_FM_ThinkPad T480s uuid=4C090A7F-2836-B211-A85C-A2222BCE7B4F
  *-core
       description: Motherboard
       product: 20L7S00R00
       vendor: LENOVO
       physical id: 0
       version: SDK0J40709 WIN
       serial: L1HF88R01EM
       slot: Not Available
     *-memory
          description: System Memory
          physical id: 3
          slot: System board or motherboard
          size: 8GiB
        *-bank:0
             description: SODIMM DDR4 Synchronous Unbuffered (Unregistered) 2400 MHz (0.4 ns)
             product: M471A1K43BB1-CRC
             vendor: Samsung
```

```
            physical id: 0
            serial: 00000000
            slot: ChannelA-DIMM0
            size: 8GiB
            width: 64 bits
            clock: 2400MHz (0.4ns)
         *-bank:1
            description: [empty]
            physical id: 1
            slot: ChannelB-DIMM0
      *-cache:0
           description: L1 cache
           physical id: 7
           slot: L1 Cache
           size: 256KiB
           capacity: 256KiB
           capabilities: synchronous internal write-back unified
           configuration: level=1
      *-cache:1
           description: L2 cache
           physical id: 8
           slot: L2 Cache
           size: 1MiB
           capacity: 1MiB
           capabilities: synchronous internal write-back unified
           configuration: level=2
      *-cache:2
           description: L3 cache
           physical id: 9
           slot: L3 Cache
           size: 6MiB
           capacity: 6MiB
           capabilities: synchronous internal write-back unified
           configuration: level=3
      *-cpu
           description: CPU
           product: Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
           vendor: Intel Corp.
           physical id: a
           bus info: cpu@0
           version: Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
           serial: None
           slot: U3E1
           size: 3123MHz
           capacity: 3400MHz
           width: 64 bits
           clock: 100MHz
           capabilities: x86-64 fpu fpu_exception wp vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
            pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp constant_tsc art
           arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf tsc_known_freq pni pclmulqdq
           dtes64 monitor ds_cpl vmx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt
           tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb invpcid_single
           pti ssbd ibrs ibpb stibp tpr_shadow vnmi flexpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1 avx2 smep
            bmi2 erms invpcid mpx rdseed adx smap clflushopt intel_pt xsaveopt xsavec xgetbv1 xsaves dtherm ida
           arat pln pts hwp hwp_notify hwp_act_window hwp_epp flush_l1d cpufreq
           configuration: cores=4 enabledcores=4 threads=8
      *-firmware
           description: BIOS
           vendor: LENOVO
           physical id: b
           version: N22ET48W (1.25 )
           date: 07/18/2018
           size: 128KiB
           capacity: 15MiB
           capabilities: pci pnp upgrade shadowing cdboot bootselect edd int13floppy720 int5printscreen
          int9keyboard int14serial int17printer int10video acpi usb biosbootspecification uefi
      *-pci
           description: Host bridge
           product: Xeon E3-1200 v6/7th Gen Core Processor Host Bridge/DRAM Registers
           vendor: Intel Corporation
           physical id: 100
           bus info: pci@0000:00:00.0
           version: 08
           width: 32 bits
           clock: 33MHz
           configuration: driver=skl_uncore
           resources: irq:0
         *-display
            description: VGA compatible controller
            product: UHD Graphics 620
            vendor: Intel Corporation
            physical id: 2
            bus info: pci@0000:00:02.0
            version: 07
            width: 64 bits
            clock: 33MHz
            capabilities: pciexpress msi pm vga_controller bus_master cap_list rom
            configuration: driver=i915 latency=0
            resources: irq:160 memory:db000000-dbffffff memory:80000000-9fffffff ioport:e000(size=64) memory
          :c0000-dffff
         *-generic:0
            description: Signal processing controller
            product: Xeon E3-1200 v5/E3-1500 v5/6th Gen Core Processor Thermal Subsystem
            vendor: Intel Corporation
            physical id: 4
            bus info: pci@0000:00:04.0
            version: 08
            width: 64 bits
            clock: 33MHz
```

```
             capabilities: msi pm cap_list
             configuration: driver=proc_thermal latency=0
             resources: irq:16 memory:dc240000-dc247fff
        *-generic:1 UNCLAIMED
             description: System peripheral
             product: Xeon E3-1200 v5/v6 / E3-1500 v5 / 6th/7th Gen Core Processor Gaussian Mixture Model
             vendor: Intel Corporation
             physical id: 8
             bus info: pci@0000:00:08.0
             version: 00
             width: 64 bits
             clock: 33MHz
             capabilities: msi pm cap_list
             configuration: latency=0
             resources: memory:dc250000-dc250fff
        *-usb
             description: USB controller
             product: Sunrise Point-LP USB 3.0 xHCI Controller
             vendor: Intel Corporation
             physical id: 14
             bus info: pci@0000:00:14.0
             version: 21
             width: 64 bits
             clock: 33MHz
             capabilities: pm msi xhci bus_master cap_list
             configuration: driver=xhci_hcd latency=0
             resources: irq:138 memory:dc220000-dc22ffff
           *-usbhost:0
                product: xHCI Host Controller
                vendor: Linux 5.2.11-arch1-1-ARCH xhci-hcd
                physical id: 0
                bus info: usb@1
                logical name: usb1
                version: 5.02
                capabilities: usb-2.00
                configuration: driver=hub slots=12 speed=480Mbit/s
              *-usb
                   description: Video
                   product: Integrated Camera
                   vendor: Chicony Electronics Co.,Ltd.
                   physical id: 8
                   bus info: usb@1:8
                   version: 0.27
                   serial: 0001
                   capabilities: usb-2.01
                   configuration: driver=uvcvideo maxpower=500mA speed=480Mbit/s
           *-usbhost:1
                product: xHCI Host Controller
                vendor: Linux 5.2.11-arch1-1-ARCH xhci-hcd
                physical id: 1
                bus info: usb@2
                logical name: usb2
                version: 5.02
                capabilities: usb-3.00
                configuration: driver=hub slots=6 speed=5000Mbit/s
              *-usb
                   description: Mass storage device
                   product: USB3.0-CRW
                   vendor: Generic
                   physical id: 3
                   bus info: usb@2:3
                   version: 2.04
                   serial: 20120501030900000
                   capabilities: usb-3.00 scsi
                   configuration: driver=usb-storage maxpower=800mA speed=5000Mbit/s
        *-generic:2
             description: Signal processing controller
             product: Sunrise Point-LP Thermal subsystem
             vendor: Intel Corporation
             physical id: 14.2
             bus info: pci@0000:00:14.2
             version: 21
             width: 64 bits
             clock: 33MHz
             capabilities: pm msi cap_list
             configuration: driver=intel_pch_thermal latency=0
             resources: irq:18 memory:dc251000-dc251fff
        *-generic:3
             description: Signal processing controller
             product: Sunrise Point-LP Serial IO I2C Controller #0
             vendor: Intel Corporation
             physical id: 15
             bus info: pci@0000:00:15.0
             version: 21
             width: 64 bits
             clock: 33MHz
             capabilities: pm bus_master cap_list
             configuration: driver=intel-lpss latency=0
             resources: irq:16 memory:dc252000-dc252fff
        *-communication
             description: Communication controller
             product: Sunrise Point-LP CSME HECI #1
             vendor: Intel Corporation
             physical id: 16
             bus info: pci@0000:00:16.0
             version: 21
             width: 64 bits
             clock: 33MHz
             capabilities: pm msi bus_master cap_list
```

```
253             configuration: driver=mei_me latency=0
254             resources: irq:129 memory:dc253000-dc253fff
255          *-pci:0
256               description: PCI bridge
257               product: Sunrise Point-LP PCI Express Root Port #1
258               vendor: Intel Corporation
259               physical id: 1c
260               bus info: pci@0000:00:1c.0
261               version: f1
262               width: 32 bits
263               clock: 33MHz
264               capabilities: pci pciexpress msi pm normal_decode bus_master cap_list
265               configuration: driver=pcieport
266               resources: irq:122 ioport:2000(size=4096) memory:7f800000-7f9fffff ioport:7fa00000(size=2097152)
267          *-pci:1
268               description: PCI bridge
269               product: Sunrise Point-LP PCI Express Root Port #5
270               vendor: Intel Corporation
271               physical id: 1c.4
272               bus info: pci@0000:00:1c.4
273               version: f1
274               width: 32 bits
275               clock: 33MHz
276               capabilities: pci pciexpress msi pm normal_decode bus_master cap_list
277               configuration: driver=pcieport
278               resources: irq:123 ioport:3000(size=8192) memory:c4000000-da0fffff ioport:a0000000(size
    =570425344)
279            *-pci
280                 description: PCI bridge
281                 product: JHL6240 Thunderbolt 3 Bridge (Low Power) [Alpine Ridge LP 2016]
282                 vendor: Intel Corporation
283                 physical id: 0
284                 bus info: pci@0000:04:00.0
285                 logical name: /dev/fb0
286                 version: 01
287                 width: 64 bits
288                 clock: 33MHz
289                 capabilities: pci pm msi pciexpress normal_decode bus_master cap_list fb
290                 configuration: depth=32 driver=pcieport mode=1920x1080 visual=truecolor xres=1920 yres=1080
291                 resources: iomemory:31310-3130f irq:16 ioport:3000(size=4096) memory:c4000000-da0fffff ioport
    :a0000000(size=570425344)
292               *-pci:0
293                    description: PCI bridge
294                    product: JHL6240 Thunderbolt 3 Bridge (Low Power) [Alpine Ridge LP 2016]
295                    vendor: Intel Corporation
296                    physical id: 0
297                    bus info: pci@0000:05:00.0
298                    version: 01
299                    width: 32 bits
300                    clock: 33MHz
301                    capabilities: pci pm msi pciexpress normal_decode bus_master cap_list
302                    configuration: driver=pcieport
303                    resources: irq:126 memory:da000000-da0fffff
304                  *-generic
305                       description: System peripheral
306                       product: JHL6240 Thunderbolt 3 NHI (Low Power) [Alpine Ridge LP 2016]
307                       vendor: Intel Corporation
308                       physical id: 0
309                       bus info: pci@0000:06:00.0
310                       version: 01
311                       width: 32 bits
312                       clock: 33MHz
313                       capabilities: pm msi pciexpress msix bus_master cap_list
314                       configuration: driver=thunderbolt latency=0
315                       resources: irq:16 memory:da000000-da03ffff memory:da040000-da040fff
316               *-pci:1
317                    description: PCI bridge
318                    product: JHL6240 Thunderbolt 3 Bridge (Low Power) [Alpine Ridge LP 2016]
319                    vendor: Intel Corporation
320                    physical id: 1
321                    bus info: pci@0000:05:01.0
322                    version: 01
323                    width: 32 bits
324                    clock: 33MHz
325                    capabilities: pci pm msi pciexpress normal_decode bus_master cap_list
326                    configuration: driver=pcieport
327                    resources: irq:127 ioport:3000(size=4096) memory:c4000000-d9efffff ioport:a0000000(size
    =570425344)
328               *-pci:2
329                    description: PCI bridge
330                    product: JHL6240 Thunderbolt 3 Bridge (Low Power) [Alpine Ridge LP 2016]
331                    vendor: Intel Corporation
332                    physical id: 2
333                    bus info: pci@0000:05:02.0
334                    version: 01
335                    width: 32 bits
336                    clock: 33MHz
337                    capabilities: pci pm msi pciexpress normal_decode bus_master cap_list
338                    configuration: driver=pcieport
339                    resources: irq:128 memory:d9f00000-d9ffffff
340                  *-usb
341                       description: USB controller
342                       product: JHL6240 Thunderbolt 3 USB 3.1 Controller (Low Power) [Alpine Ridge LP 2016]
343                       vendor: Intel Corporation
344                       physical id: 0
345                       bus info: pci@0000:3c:00.0
346                       version: 01
347                       width: 32 bits
348                       clock: 33MHz
```

```
                    capabilities: pm msi pciexpress xhci bus_master cap_list
                    configuration: driver=xhci_hcd latency=0
                    resources: irq:139 memory:d9f00000-d9f0ffff
                  *-usbhost:0
                       product: xHCI Host Controller
                       vendor: Linux 5.2.11-arch1-1-ARCH xhci-hcd
                       physical id: 0
                       bus info: usb@3
                       logical name: usb3
                       version: 5.02
                       capabilities: usb-2.00
                       configuration: driver=hub slots=2 speed=480Mbit/s
                  *-usbhost:1
                       product: xHCI Host Controller
                       vendor: Linux 5.2.11-arch1-1-ARCH xhci-hcd
                       physical id: 1
                       bus info: usb@4
                       logical name: usb4
                       version: 5.02
                       capabilities: usb-3.00
                       configuration: driver=hub slots=2 speed=5000Mbit/s
     *-pci:2
          description: PCI bridge
          product: Sunrise Point-LP PCI Express Root Port #7
          vendor: Intel Corporation
          physical id: 1c.6
          bus info: pci@0000:00:1c.6
          version: f1
          width: 32 bits
          clock: 33MHz
          capabilities: pci pciexpress msi pm normal_decode bus_master cap_list
          configuration: driver=pcieport
          resources: irq:124 memory:dc100000-dc1fffff
        *-network
             description: Wireless interface
             product: Wireless 8265 / 8275
             vendor: Intel Corporation
             physical id: 0
             bus info: pci@0000:3d:00.0
             logical name: wlp61s0
             version: 78
             serial: 04:d3:b0:c1:10:5b
             width: 64 bits
             clock: 33MHz
             capabilities: pm msi pciexpress bus_master cap_list ethernet physical wireless
             configuration: broadcast=yes driver=iwlwifi driverversion=5.2.11-arch1-1-ARCH firmware=36.77
d01142.0 ip=10.32.122.22 latency=0 link=yes multicast=yes wireless=IEEE 802.11
             resources: irq:158 memory:dc100000-dc101fff
     *-pci:3
          description: PCI bridge
          product: Sunrise Point-LP PCI Express Root Port #9
          vendor: Intel Corporation
          physical id: 1d
          bus info: pci@0000:00:1d.0
          version: f1
          width: 32 bits
          clock: 33MHz
          capabilities: pci pciexpress msi pm normal_decode bus_master cap_list
          configuration: driver=pcieport
          resources: irq:125 memory:dc000000-dc0fffff
        *-storage
             description: Non-Volatile memory controller
             product: NVMe SSD Controller SM981/PM981/PM983
             vendor: Samsung Electronics Co Ltd
             physical id: 0
             bus info: pci@0000:3e:00.0
             version: 00
             width: 64 bits
             clock: 33MHz
             capabilities: storage pm msi pciexpress msix nvm_express bus_master cap_list
             configuration: driver=nvme latency=0
             resources: irq:16 memory:dc000000-dc003fff
     *-isa
          description: ISA bridge
          product: Sunrise Point LPC Controller/eSPI Controller
          vendor: Intel Corporation
          physical id: 1f
          bus info: pci@0000:00:1f.0
          version: 21
          width: 32 bits
          clock: 33MHz
          capabilities: isa bus_master
          configuration: latency=0
     *-memory UNCLAIMED
          description: Memory controller
          product: Sunrise Point-LP PMC
          vendor: Intel Corporation
          physical id: 1f.2
          bus info: pci@0000:00:1f.2
          version: 21
          width: 32 bits
          clock: 33MHz (30.3ns)
          configuration: latency=0
          resources: memory:dc24c000-dc24ffff
     *-multimedia
          description: Audio device
          product: Sunrise Point-LP HD Audio
          vendor: Intel Corporation
          physical id: 1f.3
```

```
            bus info: pci@0000:00:1f.3
            version: 21
            width: 64 bits
            clock: 33MHz
            capabilities: pm msi bus_master cap_list
            configuration: driver=snd_hda_intel latency=64
            resources: irq:131 memory:dc248000-dc24bfff memory:dc230000-dc23ffff
       *-serial
            description: SMBus
            product: Sunrise Point-LP SMBus
            vendor: Intel Corporation
            physical id: 1f.4
            bus info: pci@0000:00:1f.4
            version: 21
            width: 64 bits
            clock: 33MHz
            configuration: driver=i801_smbus latency=0
            resources: irq:16 memory:dc254000-dc2540ff ioport:efa0(size=32)
       *-network
            description: Ethernet interface
            product: Ethernet Connection (4) I219-V
            vendor: Intel Corporation
            physical id: 1f.6
            bus info: pci@0000:00:1f.6
            logical name: enp0s31f6
            version: 21
            serial: 8c:16:45:fc:35:b6
            capacity: 1Gbit/s
            width: 32 bits
            clock: 33MHz
            capabilities: pm msi bus_master cap_list ethernet physical tp 10bt 10bt-fd 100bt 100bt-fd 1000bt
    -fd autonegotiation
            configuration: autonegotiation=on broadcast=yes driver=e1000e driverversion=3.2.6-k firmware
    =0.1-4 latency=0 link=no multicast=yes port=twisted pair
            resources: irq:161 memory:dc200000-dc21ffff
  *-battery
       product: 01AV478
       vendor: LGC
       physical id: 1
       slot: Front
       capacity: 57000mWh
       configuration: voltage=11.6V
  *-scsi
       physical id: 2
       bus info: scsi@0
       logical name: scsi0
       capabilities: scsi-host
       configuration: driver=usb-storage
```

B.2 Unix Pipe Transfer Speeds for Increasing Payload Length

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
1000	5000	0:00:00	15103.18496
2000	5000	0:00:00	14687.35037
3000	5000	0:00:00	15088.5092
4000	5000	0:00:00	14302.38678
5000	5000	0:00:00	14360.80076
6000	5000	0:00:00	13615.44536
7000	5000	0:00:00	14485.78365
8000	5000	0:00:00	14253.94834
9000	5000	0:00:00	13848.22895
10000	5000	0:00:00	12335.53646
11000	5000	0:00:00	12686.94209
12000	5000	0:00:00	12688.1011
13000	5000	0:00:00	12026.57392
14000	5000	0:00:00	12637.87926
15000	5000	0:00:00	13162.57091
16000	5000	0:00:00	11964.58483
17000	5000	0:00:00	11378.16939
18000	5000	0:00:00	11736.34535
19000	5000	0:00:00	12688.74508
20000	5000	0:00:00	12074.11575
21000	5000	0:00:00	12420.50874
22000	5000	0:00:00	12338.33694
23000	5000	0:00:00	11323.33558
24000	5000	0:00:00	10860.73497
25000	5000	0:00:00	11476.18119
26000	5000	0:00:00	11769.44135
27000	5000	0:00:00	11016.85579
28000	5000	0:00:00	11384.5166
29000	5000	0:00:00	12037.4316
30000	5000	0:00:00	11248.01191
31000	5000	0:00:00	11894.11384
32000	5000	0:00:00	11118.74598
33000	5000	0:00:00	11340.6715

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
34000	5000	0:00:00	10089.94174
35000	5000	0:00:00	11598.58683
36000	5000	0:00:00	10978.87664
37000	5000	0:00:00	10736.17985
38000	5000	0:00:00	11196.05182
39000	5000	0:00:00	10980.17858
40000	5000	0:00:00	10977.71962
41000	5000	0:00:00	11818.98981
42000	5000	0:00:00	11020.15807
43000	5000	0:00:00	10806.30137
44000	5000	0:00:00	10097.76658
45000	5000	0:00:00	11074.24618
46000	5000	0:00:00	10671.08307
47000	5000	0:00:00	9636.882276
48000	5000	0:00:00	11899.91694
49000	5000	0:00:00	11519.35713
50000	5000	0:00:00	10836.84266
51000	5000	0:00:00	10379.34428
52000	5000	0:00:00	10914.07567
53000	5000	0:00:00	10621.57454
54000	5000	0:00:00	10226.83111
55000	5000	0:00:00	10747.94983
56000	5000	0:00:00	10854.60474
57000	5000	0:00:00	10406.84521
58000	5000	0:00:00	9600.725047
59000	5000	0:00:00	10435.69006
60000	5000	0:00:00	10826.72903
61000	5000	0:00:00	10266.58207
62000	5000	0:00:00	9886.815733
63000	5000	0:00:00	5656.223769
64000	5000	0:00:00	6626.360723
65000	5000	0:00:00	8956.512549
66000	5000	0:00:00	6141.299008
67000	5000	0:00:00	9248.999721

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
68000	5000	0:00:00	7368.39313
69000	5000	0:00:00	7714.644555
70000	5000	0:00:00	7505.828276
71000	5000	0:00:00	8057.394432
72000	5000	0:00:00	8969.22122
73000	5000	0:00:00	9146.352618
74000	5000	0:00:00	8409.184175
75000	5000	0:00:00	6383.502986
76000	5000	0:00:00	8521.705636
77000	5000	0:00:00	9751.70216
78000	5000	0:00:00	6443.95499
79000	5000	0:00:00	7861.326206
80000	5000	0:00:00	9107.916054
81000	5000	0:00:00	8686.557726
82000	5000	0:00:00	9427.860837
83000	5000	0:00:00	8975.935517
84000	5000	0:00:00	8651.791353
85000	5000	0:00:00	7616.007019
86000	5000	0:00:00	7007.836162
87000	5000	0:00:00	5775.679538
88000	5000	0:00:00	8422.485341
89000	5000	0:00:00	8423.946922
90000	5000	0:00:00	5944.794263
91000	5000	0:00:00	9090.545469
92000	5000	0:00:00	8956.239812
93000	5000	0:00:00	5824.118607
94000	5000	0:00:00	6632.302497
95000	5000	0:00:00	9130.568962
96000	5000	0:00:00	9348.67763
97000	5000	0:00:00	6763.794759
98000	5000	0:00:00	5743.145843
99000	5000	0:00:00	8722.974046
100000	5000	0:00:00	8758.162608
101000	5000	0:00:00	5144.218156

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
102000	5000	0:00:00	5150.837114
103000	5000	0:00:00	9025.270758
104000	5000	0:00:00	8959.064244
105000	5000	0:00:00	5149.574542
106000	5000	0:00:00	8385.631723
107000	5000	0:00:00	8267.537514
108000	5000	0:00:00	9006.591023
109000	5000	0:00:00	8617.911122
110000	5000	0:00:00	8710.224759
111000	5000	0:00:00	8854.54114
112000	5000	0:00:00	8325.632124
113000	5000	0:00:00	5484.946017
114000	5000	0:00:00	6126.565031
115000	5000	0:00:00	8511.768371
116000	5000	0:00:01	4858.203611
117000	5000	0:00:00	8107.393781
118000	5000	0:00:00	7211.85398
119000	5000	0:00:00	8392.993529
120000	5000	0:00:00	8761.799954
121000	5000	0:00:00	8608.118833
122000	5000	0:00:00	6339.176363
123000	5000	0:00:00	5104.473254
124000	5000	0:00:00	8260.203003
125000	5000	0:00:00	8821.671671
126000	5000	0:00:00	9125.486388
127000	5000	0:00:00	8433.722746
128000	5000	0:00:00	8440.713274
129000	5000	0:00:00	8330.820203
130000	5000	0:00:00	8562.332927
131000	5000	0:00:00	9063.553638
132000	5000	0:00:00	7892.385742
133000	5000	0:00:00	7910.966815
134000	5000	0:00:00	6212.214207
135000	5000	0:00:01	4617.299729

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
136000	5000	0:00:00	5871.128724
137000	5000	0:00:00	6992.86029
138000	5000	0:00:00	6624.315211
139000	5000	0:00:01	4500.344726
140000	5000	0:00:00	8155.050341
141000	5000	0:00:00	6940.935416
142000	5000	0:00:00	7213.175297
143000	5000	0:00:00	7351.135824
144000	5000	0:00:01	3905.041878
145000	5000	0:00:01	4560.787082
146000	5000	0:00:01	2963.614519
147000	5000	0:00:01	3477.235236
148000	5000	0:00:01	4667.662435
149000	5000	0:00:00	6714.031924
150000	5000	0:00:00	7854.965909
151000	5000	0:00:00	7137.707797
152000	5000	0:00:01	4270.894283
153000	5000	0:00:01	4718.231909
154000	5000	0:00:00	7182.90698
155000	5000	0:00:00	7540.124774
156000	5000	0:00:00	7278.158102
157000	5000	0:00:00	7702.154601
158000	5000	0:00:00	7374.957594
159000	5000	0:00:00	7316.016809
160000	5000	0:00:00	7765.868387
161000	5000	0:00:00	7389.825098
162000	5000	0:00:00	7863.823737
163000	5000	0:00:00	8455.02939
164000	5000	0:00:00	8286.967749
165000	5000	0:00:00	7363.065391
166000	5000	0:00:00	7892.722121
167000	5000	0:00:00	8738.615768
168000	5000	0:00:00	7744.457679
169000	5000	0:00:00	8017.805943

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
170000	5000	0:00:00	7587.817607
171000	5000	0:00:01	4367.212714
172000	5000	0:00:00	8245.695335
173000	5000	0:00:00	5856.968152
174000	5000	0:00:00	6996.843024
175000	5000	0:00:00	7320.451291
176000	5000	0:00:01	4468.011717
177000	5000	0:00:00	5025.458975
178000	5000	0:00:00	5195.091262
179000	5000	0:00:00	5307.821819
180000	5000	0:00:01	4947.913317
181000	5000	0:00:00	5218.863478
182000	5000	0:00:01	4150.88638
183000	5000	0:00:01	4874.501095
184000	5000	0:00:00	5174.895959
185000	5000	0:00:01	4957.686149
186000	5000	0:00:00	5087.354972
187000	5000	0:00:01	4991.105849
188000	5000	0:00:00	5041.781241
189000	5000	0:00:00	5231.92605
190000	5000	0:00:01	4614.772626
191000	5000	0:00:01	4167.361227
192000	5000	0:00:01	4418.374074
193000	5000	0:00:01	4045.055446
194000	5000	0:00:01	4088.370956
195000	5000	0:00:01	4331.764657
196000	5000	0:00:01	4525.271833
197000	5000	0:00:01	3260.729921
198000	5000	0:00:01	4246.609083
199000	5000	0:00:01	4370.828045
200000	5000	0:00:01	4507.806168
201000	5000	0:00:01	3779.4409
202000	5000	0:00:00	5288.447809
203000	5000	0:00:01	4597.168512

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
204000	5000	0:00:01	3657.502672
205000	5000	0:00:01	3166.948208
206000	5000	0:00:00	6533.57342
207000	5000	0:00:00	5261.418857
208000	5000	0:00:00	6339.706852
209000	5000	0:00:00	6178.461143
210000	5000	0:00:00	5110.755175
211000	5000	0:00:01	3392.245327
212000	5000	0:00:01	3426.692426
213000	5000	0:00:01	4974.807574
214000	5000	0:00:01	3466.127615
215000	5000	0:00:01	4791.479599
216000	5000	0:00:00	6297.284737
217000	5000	0:00:00	6981.845805
218000	5000	0:00:00	5807.868733
219000	5000	0:00:00	6301.030471
220000	5000	0:00:00	6081.238043
221000	5000	0:00:00	6548.428246
222000	5000	0:00:00	6769.820681
223000	5000	0:00:00	6296.047719
224000	5000	0:00:01	4794.56718
225000	5000	0:00:01	4330.85291
226000	5000	0:00:01	4948.677268
227000	5000	0:00:01	4548.630316
228000	5000	0:00:00	5577.163968
229000	5000	0:00:00	6902.282999
230000	5000	0:00:01	3279.854217
231000	5000	0:00:00	6483.839678
232000	5000	0:00:00	6510.509916
233000	5000	0:00:00	6420.422079
234000	5000	0:00:00	6861.214106
235000	5000	0:00:01	3630.602201
236000	5000	0:00:00	6213.387613
237000	5000	0:00:00	6505.435942

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
238000	5000	0:00:00	6805.128345
239000	5000	0:00:00	6532.096764
240000	5000	0:00:00	6747.101108
241000	5000	0:00:00	5981.698396
242000	5000	0:00:00	6202.904448
243000	5000	0:00:00	7668.535233
244000	5000	0:00:00	7781.519824
245000	5000	0:00:01	4464.353477
246000	5000	0:00:01	4013.092312
247000	5000	0:00:01	4712.268863
248000	5000	0:00:01	4721.600283
249000	5000	0:00:01	4472.844022
250000	5000	0:00:01	4655.346106
251000	5000	0:00:01	4724.442185
252000	5000	0:00:01	4746.291722
253000	5000	0:00:01	4635.741941
254000	5000	0:00:01	4606.745196
255000	5000	0:00:01	4700.653015
256000	5000	0:00:01	4634.470076
257000	5000	0:00:01	4630.74015
258000	5000	0:00:01	4594.059697
259000	5000	0:00:01	4619.24915
260000	5000	0:00:01	4666.224931
261000	5000	0:00:01	4636.365238
262000	5000	0:00:01	4634.848125
263000	5000	0:00:01	4296.50342
264000	5000	0:00:01	4380.040331
265000	5000	0:00:01	4383.680784
266000	5000	0:00:01	4424.117101
267000	5000	0:00:01	4197.440904
268000	5000	0:00:01	4221.265214
269000	5000	0:00:01	4178.617513
270000	5000	0:00:01	4318.255425
271000	5000	0:00:01	4200.448776

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
272000	5000	0:00:01	4259.023593
273000	5000	0:00:01	4265.691132
274000	5000	0:00:01	4378.352724
275000	5000	0:00:01	4167.340387
276000	5000	0:00:01	4138.740515
277000	5000	0:00:01	4164.577437
278000	5000	0:00:01	4192.225267
279000	5000	0:00:01	4071.783922
280000	5000	0:00:01	4215.42153
281000	5000	0:00:01	3704.194853
282000	5000	0:00:01	4001.73195
283000	5000	0:00:01	4045.922843
284000	5000	0:00:01	3955.280022
285000	5000	0:00:01	4018.578693
286000	5000	0:00:01	4048.871499
287000	5000	0:00:01	3994.535475
288000	5000	0:00:01	3989.286373
289000	5000	0:00:01	3907.678745
290000	5000	0:00:01	4000.579284
291000	5000	0:00:01	3922.697749
292000	5000	0:00:01	3924.341829
293000	5000	0:00:01	3870.097844
294000	5000	0:00:01	3820.124108
295000	5000	0:00:01	3936.074993
296000	5000	0:00:01	3944.110378
297000	5000	0:00:01	3941.936847
298000	5000	0:00:01	3888.17908
299000	5000	0:00:01	3894.490464
300000	5000	0:00:01	4013.427322
301000	5000	0:00:01	3882.16541
302000	5000	0:00:01	3905.694659
303000	5000	0:00:01	3888.236529
304000	5000	0:00:01	3865.242196
305000	5000	0:00:01	3937.572156

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
306000	5000	0:00:01	3878.693098
307000	5000	0:00:01	3858.316447
308000	5000	0:00:01	3884.973699
309000	5000	0:00:01	3809.886656
310000	5000	0:00:01	3944.166381
311000	5000	0:00:01	3884.620554
312000	5000	0:00:01	3858.096138
313000	5000	0:00:01	3839.753273
314000	5000	0:00:01	3859.236658
315000	5000	0:00:01	3805.64133
316000	5000	0:00:01	3787.181602
317000	5000	0:00:01	3792.803686
318000	5000	0:00:01	3704.859071
319000	5000	0:00:01	3783.367409
320000	5000	0:00:01	3751.131904
321000	5000	0:00:01	3759.183686
322000	5000	0:00:01	3789.563996
323000	5000	0:00:01	3760.190115
324000	5000	0:00:01	3695.652656
325000	5000	0:00:01	3731.738736
326000	5000	0:00:01	3766.759254
327000	5000	0:00:01	3757.355964
328000	5000	0:00:01	3571.160734
329000	5000	0:00:01	3610.199536
330000	5000	0:00:01	3569.985277
331000	5000	0:00:01	3514.787767
332000	5000	0:00:01	3455.312785
333000	5000	0:00:01	3492.903119
334000	5000	0:00:01	3509.032953
335000	5000	0:00:01	3451.496465
336000	5000	0:00:01	3473.459642
337000	5000	0:00:01	3506.638417
338000	5000	0:00:01	3503.078505
339000	5000	0:00:01	3526.391514

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
340000	5000	0:00:01	3518.2725
341000	5000	0:00:01	3492.373704
342000	5000	0:00:01	3460.334531
343000	5000	0:00:01	3303.714432
344000	5000	0:00:01	3463.947237
345000	5000	0:00:01	3491.63718
346000	5000	0:00:01	3440.966774
347000	5000	0:00:01	3465.426137
348000	5000	0:00:01	3514.466599
349000	5000	0:00:01	3457.513384
350000	5000	0:00:01	3445.816262
351000	5000	0:00:01	3474.881472
352000	5000	0:00:01	3405.354307
353000	5000	0:00:01	3398.604669
354000	5000	0:00:01	3373.634774
355000	5000	0:00:01	3418.263508
356000	5000	0:00:01	3423.285875
357000	5000	0:00:01	3380.303781
358000	5000	0:00:01	3379.442446
359000	5000	0:00:01	3375.484129
360000	5000	0:00:01	3394.55066
361000	5000	0:00:01	3377.118804
362000	5000	0:00:01	3410.420198
363000	5000	0:00:01	3407.29898
364000	5000	0:00:01	3361.575046
365000	5000	0:00:01	3354.40346
366000	5000	0:00:01	3302.771686
367000	5000	0:00:01	3277.579521
368000	5000	0:00:01	3346.838944
369000	5000	0:00:01	3243.966385
370000	5000	0:00:01	3253.630238
371000	5000	0:00:01	3291.387886
372000	5000	0:00:01	3230.416569
373000	5000	0:00:01	3222.891294

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
374000	5000	0:00:01	3259.613905
375000	5000	0:00:01	3201.745848
376000	5000	0:00:01	3212.042332
377000	5000	0:00:01	3181.507551
378000	5000	0:00:01	3211.838064
379000	5000	0:00:01	3218.399979
380000	5000	0:00:01	3240.549908
381000	5000	0:00:01	3225.600429
382000	5000	0:00:01	3155.151416
383000	5000	0:00:01	3035.909343
384000	5000	0:00:01	3135.271935
385000	5000	0:00:01	3148.196304
386000	5000	0:00:01	3115.792832
387000	5000	0:00:01	3134.821789
388000	5000	0:00:01	3129.00121
389000	5000	0:00:01	3154.476612
390000	5000	0:00:01	3105.235178
391000	5000	0:00:01	3184.857404
392000	5000	0:00:01	3089.192399
393000	5000	0:00:01	3123.851984
394000	5000	0:00:01	3004.990087
395000	5000	0:00:01	3024.547225
396000	5000	0:00:01	3073.775531
397000	5000	0:00:01	3053.63092
398000	5000	0:00:01	2958.933553
399000	5000	0:00:01	2939.248669
400000	5000	0:00:01	2962.188844
401000	5000	0:00:01	2990.160578
402000	5000	0:00:01	3005.759637
403000	5000	0:00:01	3029.151953
404000	5000	0:00:01	3012.284095
405000	5000	0:00:01	2940.512261
406000	5000	0:00:02	2002.244115
407000	5000	0:00:02	1849.704824

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
408000	5000	0:00:02	2138.475712
409000	5000	0:00:01	3009.191274
410000	5000	0:00:01	2968.740352
411000	5000	0:00:01	2959.726995
412000	5000	0:00:01	2966.674162
413000	5000	0:00:01	2971.249006
414000	5000	0:00:01	2976.668869
415000	5000	0:00:01	2999.646642
416000	5000	0:00:01	2993.108667
417000	5000	0:00:01	2913.35394
418000	5000	0:00:01	2965.929772
419000	5000	0:00:01	2973.804352
420000	5000	0:00:01	2946.737137
421000	5000	0:00:01	2923.255197
422000	5000	0:00:01	2906.632004
423000	5000	0:00:01	2943.198621
424000	5000	0:00:01	2607.915545
425000	5000	0:00:01	2805.617295
426000	5000	0:00:01	2923.197089
427000	5000	0:00:01	2948.16188
428000	5000	0:00:01	2882.43582
429000	5000	0:00:01	2974.949143
430000	5000	0:00:01	2502.032902
431000	5000	0:00:01	2914.503619
432000	5000	0:00:01	2855.110019
433000	5000	0:00:01	2864.587511
434000	5000	0:00:01	2776.523715
435000	5000	0:00:01	2827.696039
436000	5000	0:00:01	2767.308129
437000	5000	0:00:01	2737.032487
438000	5000	0:00:01	2794.515372
439000	5000	0:00:01	2772.531532
440000	5000	0:00:01	2806.40624
441000	5000	0:00:01	2799.730778

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
442000	5000	0:00:01	2759.945463
443000	5000	0:00:01	2777.217706
444000	5000	0:00:01	2652.82247
445000	5000	0:00:01	2768.452706
446000	5000	0:00:01	2778.018539
447000	5000	0:00:01	2707.014904
448000	5000	0:00:01	2774.028216
449000	5000	0:00:01	2778.558862
450000	5000	0:00:01	2748.926269
451000	5000	0:00:01	2755.047109
452000	5000	0:00:01	2731.547848
453000	5000	0:00:01	2758.421322
454000	5000	0:00:01	2767.735511
455000	5000	0:00:01	2769.743981
456000	5000	0:00:01	2743.52487
457000	5000	0:00:01	2776.405
458000	5000	0:00:01	2735.530548
459000	5000	0:00:01	2591.781461
460000	5000	0:00:01	2615.592173
461000	5000	0:00:01	2595.320948
462000	5000	0:00:01	2641.226374
463000	5000	0:00:01	2608.300551
464000	5000	0:00:01	2575.262033
465000	5000	0:00:01	2593.689243
466000	5000	0:00:01	2631.609419
467000	5000	0:00:01	2629.682808
468000	5000	0:00:01	2602.736517
469000	5000	0:00:01	2589.339998
470000	5000	0:00:01	2598.689117
471000	5000	0:00:01	2591.414747
472000	5000	0:00:01	2592.653664
473000	5000	0:00:01	2616.618774
474000	5000	0:00:01	2585.602846
475000	5000	0:00:01	2602.300329

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
476000	5000	0:00:02	2489.535239
477000	5000	0:00:01	2536.286653
478000	5000	0:00:01	2577.510908
479000	5000	0:00:01	2573.217031
480000	5000	0:00:01	2555.583951
481000	5000	0:00:01	2570.440347
482000	5000	0:00:01	2568.154981
483000	5000	0:00:01	2555.586563
484000	5000	0:00:01	2559.927912
485000	5000	0:00:01	2545.533226
486000	5000	0:00:01	2557.275295
487000	5000	0:00:01	2557.29099
488000	5000	0:00:01	2546.042635
489000	5000	0:00:01	2549.739034
490000	5000	0:00:01	2535.256545
491000	5000	0:00:01	2509.594179
492000	5000	0:00:02	2499.690038
493000	5000	0:00:02	2492.442913
494000	5000	0:00:01	2518.583367
495000	5000	0:00:02	2480.748154
496000	5000	0:00:02	2426.957086
497000	5000	0:00:02	2453.485593
498000	5000	0:00:02	2463.877098
499000	5000	0:00:02	2443.589731
500000	5000	0:00:02	2299.491261
501000	5000	0:00:02	2394.029482
502000	5000	0:00:02	2464.289974
503000	5000	0:00:02	2380.766454
504000	5000	0:00:02	2382.276246
505000	5000	0:00:02	2397.58247
506000	5000	0:00:02	2373.045619
507000	5000	0:00:02	2384.88062
508000	5000	0:00:02	2416.801605
509000	5000	0:00:02	2423.641949

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
510000	5000	0:00:02	2412.219338
511000	5000	0:00:02	2368.86216
512000	5000	0:00:02	2328.997981
513000	5000	0:00:02	2386.191776
514000	5000	0:00:02	2416.142927
515000	5000	0:00:02	2402.112706
516000	5000	0:00:02	2407.570944
517000	5000	0:00:02	2410.523768
518000	5000	0:00:02	2421.142188
519000	5000	0:00:02	2411.313109
520000	5000	0:00:02	2426.006793
521000	5000	0:00:02	2441.31923
522000	5000	0:00:02	2479.799553
523000	5000	0:00:02	2403.702856
524000	5000	0:00:02	2365.055827
525000	5000	0:00:02	2362.279127
526000	5000	0:00:02	2391.337619
527000	5000	0:00:02	2347.569936
528000	5000	0:00:02	2352.369966
529000	5000	0:00:02	2298.623354
530000	5000	0:00:02	2339.138608
531000	5000	0:00:02	2342.639077
532000	5000	0:00:02	2298.241937
533000	5000	0:00:02	2302.697933
534000	5000	0:00:02	2278.566618
535000	5000	0:00:02	2234.377234
536000	5000	0:00:02	2309.512559
537000	5000	0:00:02	2272.720041
538000	5000	0:00:02	2314.002771
539000	5000	0:00:02	2289.217967
540000	5000	0:00:02	2280.485068
541000	5000	0:00:02	2310.149596
542000	5000	0:00:02	2256.558009
543000	5000	0:00:02	2268.876371

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
544000	5000	0:00:02	2268.701359
545000	5000	0:00:02	2221.290268
546000	5000	0:00:02	2208.123598
547000	5000	0:00:02	2239.117107
548000	5000	0:00:02	2223.566244
549000	5000	0:00:02	2218.826924
550000	5000	0:00:02	2203.66443
551000	5000	0:00:02	2226.093646
552000	5000	0:00:02	2190.762867
553000	5000	0:00:02	2183.903929
554000	5000	0:00:02	2194.247035
555000	5000	0:00:02	2203.153682
556000	5000	0:00:02	2206.427146
557000	5000	0:00:02	2198.472853
558000	5000	0:00:02	2180.078875
559000	5000	0:00:02	2200.476623
560000	5000	0:00:02	2189.764079
561000	5000	0:00:02	2162.348241
562000	5000	0:00:02	2175.074344
563000	5000	0:00:02	2152.541118
564000	5000	0:00:02	2178.495461
565000	5000	0:00:02	2187.639052
566000	5000	0:00:02	2177.172176
567000	5000	0:00:02	2151.303776
568000	5000	0:00:02	2239.965737
569000	5000	0:00:02	2167.748156
570000	5000	0:00:02	2228.495685
571000	5000	0:00:02	2192.315146
572000	5000	0:00:02	2209.170443
573000	5000	0:00:02	2210.012151
574000	5000	0:00:02	2195.344377
575000	5000	0:00:02	2212.544063
576000	5000	0:00:02	2201.331101
577000	5000	0:00:02	2216.012463

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
578000	5000	0:00:02	2218.813139
579000	5000	0:00:02	2202.523828
580000	5000	0:00:02	2202.648023
581000	5000	0:00:02	2208.616165
582000	5000	0:00:02	2187.685953
583000	5000	0:00:02	2185.975828
584000	5000	0:00:02	2198.469953
585000	5000	0:00:02	2213.563745
586000	5000	0:00:02	2184.394337
587000	5000	0:00:02	2182.132091
588000	5000	0:00:02	2219.240549
589000	5000	0:00:02	2204.230801
590000	5000	0:00:02	2095.284311
591000	5000	0:00:02	2090.140223
592000	5000	0:00:02	2078.122441
593000	5000	0:00:02	2083.649353
594000	5000	0:00:02	2032.777315
595000	5000	0:00:02	2039.160029
596000	5000	0:00:02	2019.744212
597000	5000	0:00:02	1996.168156
598000	5000	0:00:02	2024.890767
599000	5000	0:00:02	2008.907496
600000	5000	0:00:02	2017.100168
601000	5000	0:00:02	2005.684109
602000	5000	0:00:02	2023.58611
603000	5000	0:00:02	2017.045649
604000	5000	0:00:02	2015.636502
605000	5000	0:00:02	1999.996
606000	5000	0:00:02	1997.765699
607000	5000	0:00:02	1969.882079
608000	5000	0:00:02	2008.761414
609000	5000	0:00:02	1987.380926
610000	5000	0:00:02	1979.619422
611000	5000	0:00:02	1977.266966

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
612000	5000	0:00:02	1970.795957
613000	5000	0:00:02	1961.889127
614000	5000	0:00:02	2001.14946
615000	5000	0:00:02	1995.713208
616000	5000	0:00:02	2012.44091
617000	5000	0:00:02	2025.129426
618000	5000	0:00:02	2005.607679
619000	5000	0:00:02	2027.73372
620000	5000	0:00:02	2027.272491
621000	5000	0:00:02	2026.108433
622000	5000	0:00:02	2027.292219
623000	5000	0:00:02	2027.068664
624000	5000	0:00:02	2014.024052
625000	5000	0:00:02	2026.284147
626000	5000	0:00:02	2000.438496
627000	5000	0:00:02	1999.628869
628000	5000	0:00:02	2000.233627
629000	5000	0:00:02	2023.26594
630000	5000	0:00:02	1999.948801
631000	5000	0:00:02	2013.59661
632000	5000	0:00:02	1999.396982
633000	5000	0:00:02	2006.459193
634000	5000	0:00:02	1992.146163
635000	5000	0:00:02	1991.182249
636000	5000	0:00:02	1992.77023
637000	5000	0:00:02	1964.892866
638000	5000	0:00:02	1951.782387
639000	5000	0:00:02	1939.65952
640000	5000	0:00:02	1960.204708
641000	5000	0:00:02	1944.947105
642000	5000	0:00:02	1899.836804
643000	5000	0:00:02	1937.915392
644000	5000	0:00:02	1936.847912
645000	5000	0:00:02	1944.252072

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
646000	5000	0:00:02	1898.891617
647000	5000	0:00:02	1921.18308
648000	5000	0:00:02	1900.220198
649000	5000	0:00:02	1928.031965
650000	5000	0:00:02	1935.018218
651000	5000	0:00:02	1896.195436
652000	5000	0:00:02	1930.645797
653000	5000	0:00:02	1939.232971
654000	5000	0:00:02	1916.920659
655000	5000	0:00:02	1904.529734
656000	5000	0:00:02	1853.404593
657000	5000	0:00:02	1878.934253
658000	5000	0:00:02	1891.623588
659000	5000	0:00:02	1872.204798
660000	5000	0:00:02	1847.226943
661000	5000	0:00:02	1863.136941
662000	5000	0:00:02	1859.940527
663000	5000	0:00:02	1876.269061
664000	5000	0:00:02	1870.121554
665000	5000	0:00:02	1862.825273
666000	5000	0:00:02	1872.035865
667000	5000	0:00:02	1864.504935
668000	5000	0:00:02	1857.764359
669000	5000	0:00:02	1873.33133
670000	5000	0:00:02	1853.547505
671000	5000	0:00:02	1874.842279
672000	5000	0:00:02	1865.187946
673000	5000	0:00:02	1851.79767
674000	5000	0:00:02	1841.966631
675000	5000	0:00:02	1829.917914
676000	5000	0:00:02	1836.699084
677000	5000	0:00:02	1829.186872
678000	5000	0:00:02	1838.331942
679000	5000	0:00:02	1847.92467

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
680000	5000	0:00:02	1833.784322
681000	5000	0:00:02	1797.952564
682000	5000	0:00:02	1809.293254
683000	5000	0:00:02	1784.558784
684000	5000	0:00:02	1768.451671
685000	5000	0:00:02	1775.563768
686000	5000	0:00:02	1789.019499
687000	5000	0:00:02	1798.262303
688000	5000	0:00:02	1783.489383
689000	5000	0:00:02	1768.212143
690000	5000	0:00:02	1784.271575
691000	5000	0:00:02	1783.766158
692000	5000	0:00:02	1777.5433
693000	5000	0:00:02	1774.233726
694000	5000	0:00:02	1766.629014
695000	5000	0:00:02	1784.558784
696000	5000	0:00:02	1765.493085
697000	5000	0:00:02	1765.552309
698000	5000	0:00:02	1793.818501
699000	5000	0:00:02	1794.226609
700000	5000	0:00:02	1806.121524
701000	5000	0:00:02	1790.744359
702000	5000	0:00:02	1808.292104
703000	5000	0:00:02	1796.65011
704000	5000	0:00:02	1829.999623
705000	5000	0:00:02	1797.480077
706000	5000	0:00:02	1808.69832
707000	5000	0:00:02	1803.258705
708000	5000	0:00:02	1819.524296
709000	5000	0:00:02	1809.301765
710000	5000	0:00:02	1818.329268
711000	5000	0:00:02	1798.310811
712000	5000	0:00:02	1791.725525
713000	5000	0:00:02	1807.213021

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
714000	5000	0:00:02	1806.323795
715000	5000	0:00:02	1807.950135
716000	5000	0:00:02	1788.140409
717000	5000	0:00:02	1791.765333
718000	5000	0:00:02	1789.378037
719000	5000	0:00:02	1774.485594
720000	5000	0:00:02	1770.03619
721000	5000	0:00:02	1709.30295
722000	5000	0:00:02	1669.882861
723000	5000	0:00:02	1683.21208
724000	5000	0:00:02	1672.358259
725000	5000	0:00:03	1665.60734
726000	5000	0:00:03	1664.292276
727000	5000	0:00:02	1681.829481
728000	5000	0:00:02	1678.920548
729000	5000	0:00:03	1659.027949
730000	5000	0:00:03	1650.295188
731000	5000	0:00:03	1666.380605
732000	5000	0:00:03	1665.524117
733000	5000	0:00:03	1648.325203
734000	5000	0:00:03	1657.012909
735000	5000	0:00:03	1661.595477
736000	5000	0:00:02	1669.906843
737000	5000	0:00:02	1685.245373
738000	5000	0:00:02	1672.107146
739000	5000	0:00:02	1680.372344
740000	5000	0:00:02	1706.350696
741000	5000	0:00:02	1709.491713
742000	5000	0:00:02	1705.460783
743000	5000	0:00:02	1702.349617
744000	5000	0:00:02	1702.923613
745000	5000	0:00:02	1708.67326
746000	5000	0:00:02	1703.019317
747000	5000	0:00:02	1710.839985

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
748000	5000	0:00:02	1695.503795
749000	5000	0:00:02	1696.61129
750000	5000	0:00:02	1696.754076
751000	5000	0:00:02	1690.651374
752000	5000	0:00:02	1688.626694
753000	5000	0:00:02	1696.949292
754000	5000	0:00:02	1692.330864
755000	5000	0:00:02	1695.228441
756000	5000	0:00:02	1692.067419
757000	5000	0:00:02	1672.329173
758000	5000	0:00:03	1663.442914
759000	5000	0:00:02	1667.070653
760000	5000	0:00:03	1646.39461
761000	5000	0:00:03	1621.151751
762000	5000	0:00:03	1616.019537
763000	5000	0:00:03	1632.310876
764000	5000	0:00:03	1642.000535
765000	5000	0:00:03	1646.914128
766000	5000	0:00:03	1632.484082
767000	5000	0:00:03	1646.818117
768000	5000	0:00:03	1649.519907
769000	5000	0:00:03	1630.695162
770000	5000	0:00:03	1648.634996
771000	5000	0:00:03	1646.105709
772000	5000	0:00:03	1640.695012
773000	5000	0:00:03	1652.587175
774000	5000	0:00:03	1633.031658
775000	5000	0:00:03	1658.759361
776000	5000	0:00:03	1651.528209
777000	5000	0:00:03	1644.474484
778000	5000	0:00:03	1637.721039
779000	5000	0:00:02	1676.634257
780000	5000	0:00:02	1667.949876
781000	5000	0:00:03	1648.325203

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
782000	5000	0:00:03	1653.744309
783000	5000	0:00:03	1641.701315
784000	5000	0:00:03	1657.200736
785000	5000	0:00:03	1655.519502
786000	5000	0:00:03	1647.91993
787000	5000	0:00:03	1624.552314
788000	5000	0:00:03	1641.162455
789000	5000	0:00:03	1627.677285
790000	5000	0:00:03	1628.788113
791000	5000	0:00:03	1605.172508
792000	5000	0:00:03	1601.721145
793000	5000	0:00:03	1595.139546
794000	5000	0:00:03	1602.000322
795000	5000	0:00:03	1572.07393
796000	5000	0:00:03	1573.20517
797000	5000	0:00:03	1567.804569
798000	5000	0:00:03	1547.866375
799000	5000	0:00:03	1554.89172
800000	5000	0:00:03	1568.368147
801000	5000	0:00:03	1563.716767
802000	5000	0:00:03	1569.154687
803000	5000	0:00:03	1543.17606
804000	5000	0:00:03	1554.957001
805000	5000	0:00:03	1557.590669
806000	5000	0:00:03	1559.189159
807000	5000	0:00:03	1556.678024
808000	5000	0:00:03	1560.41733
809000	5000	0:00:03	1540.243956
810000	5000	0:00:03	1551.506125
811000	5000	0:00:03	1560.169497
812000	5000	0:00:03	1566.377764
813000	5000	0:00:03	1568.690937
814000	5000	0:00:03	1586.980665
815000	5000	0:00:03	1594.396907

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
816000	5000	0:00:03	1582.31203
817000	5000	0:00:03	1579.803289
818000	5000	0:00:03	1578.812089
819000	5000	0:00:03	1576.914974
820000	5000	0:00:03	1585.60675
821000	5000	0:00:03	1575.526651
822000	5000	0:00:03	1575.484454
823000	5000	0:00:03	1578.605725
824000	5000	0:00:03	1573.001753
825000	5000	0:00:03	1565.353836
826000	5000	0:00:03	1583.561366
827000	5000	0:00:03	1577.301496
828000	5000	0:00:03	1567.297891
829000	5000	0:00:03	1558.323693
830000	5000	0:00:03	1555.414117
831000	5000	0:00:03	1543.561466
832000	5000	0:00:03	1533.960662
833000	5000	0:00:03	1539.345833
834000	5000	0:00:03	1544.365291
835000	5000	0:00:03	1534.279329
836000	5000	0:00:03	1526.808
837000	5000	0:00:03	1538.224888
838000	5000	0:00:03	1533.351468
839000	5000	0:00:03	1531.109387
840000	5000	0:00:03	1510.18575
841000	5000	0:00:03	1528.478925
842000	5000	0:00:03	1522.095036
843000	5000	0:00:03	1516.456587
844000	5000	0:00:03	1524.390709
845000	5000	0:00:03	1524.371654
846000	5000	0:00:03	1527.734025
847000	5000	0:00:03	1535.142325
848000	5000	0:00:03	1538.845066
849000	5000	0:00:03	1527.384476

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
850000	5000	0:00:03	1544.085812
851000	5000	0:00:03	1542.38727
852000	5000	0:00:03	1493.732299
853000	5000	0:00:03	1493.038261
854000	5000	0:00:03	1506.429743
855000	5000	0:00:03	1501.548848
856000	5000	0:00:03	1494.576183
857000	5000	0:00:03	1486.337732
858000	5000	0:00:03	1485.737514
859000	5000	0:00:03	1483.591038
860000	5000	0:00:03	1499.82827
861000	5000	0:00:03	1474.635242
862000	5000	0:00:03	1478.374486
863000	5000	0:00:03	1473.456709
864000	5000	0:00:03	1449.657316
865000	5000	0:00:03	1450.026013
866000	5000	0:00:03	1445.984804
867000	5000	0:00:03	1435.641759
868000	5000	0:00:03	1437.236321
869000	5000	0:00:03	1446.807821
870000	5000	0:00:03	1456.544151
871000	5000	0:00:03	1444.051245
872000	5000	0:00:03	1445.819226
873000	5000	0:00:03	1437.80708
874000	5000	0:00:03	1447.292781
875000	5000	0:00:03	1447.287335
876000	5000	0:00:03	1430.671655
877000	5000	0:00:03	1439.844174
878000	5000	0:00:03	1434.265051
879000	5000	0:00:03	1447.943255
880000	5000	0:00:03	1449.741381
881000	5000	0:00:03	1458.23229
882000	5000	0:00:03	1457.4009
883000	5000	0:00:03	1469.586323

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
884000	5000	0:00:03	1462.556648
885000	5000	0:00:03	1465.133629
886000	5000	0:00:03	1464.374978
887000	5000	0:00:03	1467.773995
888000	5000	0:00:03	1445.570094
889000	5000	0:00:03	1461.4251
890000	5000	0:00:03	1451.312669
891000	5000	0:00:03	1458.198268
892000	5000	0:00:03	1450.265747
893000	5000	0:00:03	1452.426918
894000	5000	0:00:03	1458.753029
895000	5000	0:00:03	1461.412285
896000	5000	0:00:03	1430.807167
897000	5000	0:00:03	1433.807694
898000	5000	0:00:03	1436.435296
899000	5000	0:00:03	1426.954078
900000	5000	0:00:03	1412.663625
901000	5000	0:00:03	1413.547439
902000	5000	0:00:03	1427.580688
903000	5000	0:00:03	1419.73016
904000	5000	0:00:03	1414.090335
905000	5000	0:00:03	1404.800597
906000	5000	0:00:03	1376.728931
907000	5000	0:00:03	1413.521863
908000	5000	0:00:03	1413.150325
909000	5000	0:00:03	1398.712122
910000	5000	0:00:03	1413.469916
911000	5000	0:00:03	1422.955633
912000	5000	0:00:03	1409.536643
913000	5000	0:00:03	1411.530624
914000	5000	0:00:03	1426.556316
915000	5000	0:00:03	1433.97958
916000	5000	0:00:03	1417.150813
917000	5000	0:00:03	1414.497582

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
918000	5000	0:00:03	1408.18771
919000	5000	0:00:03	1405.331265
920000	5000	0:00:03	1404.231286
921000	5000	0:00:03	1417.348459
922000	5000	0:00:03	1394.860441
923000	5000	0:00:03	1405.935466
924000	5000	0:00:03	1391.846563
925000	5000	0:00:03	1383.912953
926000	5000	0:00:03	1390.153542
927000	5000	0:00:03	1395.074494
928000	5000	0:00:03	1384.133238
929000	5000	0:00:03	1372.976096
930000	5000	0:00:03	1373.549017
931000	5000	0:00:03	1363.668968
932000	5000	0:00:03	1358.358646
933000	5000	0:00:03	1359.188358
934000	5000	0:00:03	1358.271561
935000	5000	0:00:03	1358.816395
936000	5000	0:00:03	1355.341183
937000	5000	0:00:03	1354.355091
938000	5000	0:00:03	1354.32501
939000	5000	0:00:03	1357.203098
940000	5000	0:00:03	1357.536582
941000	5000	0:00:03	1364.265414
942000	5000	0:00:03	1371.980751
943000	5000	0:00:03	1364.832205
944000	5000	0:00:03	1374.08201
945000	5000	0:00:03	1365.028569
946000	5000	0:00:03	1377.244668
947000	5000	0:00:03	1366.813708
948000	5000	0:00:03	1371.502429
949000	5000	0:00:03	1375.269897
950000	5000	0:00:03	1378.2906
951000	5000	0:00:03	1364.003405

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
952000	5000	0:00:03	1380.629655
953000	5000	0:00:03	1371.022185
954000	5000	0:00:03	1375.575231
955000	5000	0:00:03	1369.25079
956000	5000	0:00:03	1369.841997
957000	5000	0:00:03	1358.758421
958000	5000	0:00:03	1356.98062
959000	5000	0:00:03	1347.151153
960000	5000	0:00:03	1347.694001
961000	5000	0:00:03	1344.236706
962000	5000	0:00:03	1340.407468
963000	5000	0:00:03	1339.328884
964000	5000	0:00:03	1341.925233
965000	5000	0:00:03	1339.968141
966000	5000	0:00:03	1337.941919
967000	5000	0:00:03	1335.128815
968000	5000	0:00:03	1330.751675
969000	5000	0:00:03	1338.083351
970000	5000	0:00:03	1334.027383
971000	5000	0:00:03	1339.869036
972000	5000	0:00:03	1337.861728
973000	5000	0:00:03	1344.805058
974000	5000	0:00:03	1351.03112
975000	5000	0:00:03	1324.952216
976000	5000	0:00:03	1339.271126
977000	5000	0:00:03	1343.004758
978000	5000	0:00:03	1352.618426
979000	5000	0:00:03	1355.724846
980000	5000	0:00:03	1356.150291
981000	5000	0:00:03	1355.784768
982000	5000	0:00:03	1353.757381
983000	5000	0:00:03	1355.905729
984000	5000	0:00:03	1321.934032
985000	5000	0:00:03	1319.559943

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
986000	5000	0:00:03	1315.816483
987000	5000	0:00:03	1307.752987
988000	5000	0:00:03	1300.317069
989000	5000	0:00:03	1287.228913
990000	5000	0:00:03	1286.680031
991000	5000	0:00:03	1271.833243
992000	5000	0:00:03	1270.21677
993000	5000	0:00:03	1273.505408
994000	5000	0:00:03	1277.40872
995000	5000	0:00:03	1276.995363
996000	5000	0:00:03	1278.845315
997000	5000	0:00:03	1269.82321
998000	5000	0:00:03	1277.778587
999000	5000	0:00:03	1274.473069
1000000	5000	0:00:03	1264.268215
1001000	5000	0:00:03	1276.60574
1002000	5000	0:00:03	1274.904299
1003000	5000	0:00:03	1280.094379
1004000	5000	0:00:03	1285.769133
1005000	5000	0:00:03	1283.733302
1006000	5000	0:00:03	1298.24944
1007000	5000	0:00:03	1284.746945
1008000	5000	0:00:03	1275.794654
1009000	5000	0:00:03	1289.265217
1010000	5000	0:00:03	1290.920235
1011000	5000	0:00:03	1278.846951
1012000	5000	0:00:03	1286.17992
1013000	5000	0:00:03	1282.285381
1014000	5000	0:00:03	1284.365115
1015000	5000	0:00:03	1285.965563
1016000	5000	0:00:03	1285.732102
1017000	5000	0:00:03	1280.295308
1018000	5000	0:00:03	1262.603944
1019000	5000	0:00:03	1263.489966

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
1020000	5000	0:00:03	1257.09378
1021000	5000	0:00:03	1257.507951
1022000	5000	0:00:03	1252.061519
1023000	5000	0:00:03	1259.436009
1024000	5000	0:00:03	1250.226916
1025000	5000	0:00:04	1243.337576
1026000	5000	0:00:04	1246.469065
1027000	5000	0:00:04	1244.655759
1028000	5000	0:00:04	1249.793159
1029000	5000	0:00:03	1255.628354
1030000	5000	0:00:03	1264.315848
1031000	5000	0:00:03	1254.787642
1032000	5000	0:00:03	1257.740766
1033000	5000	0:00:03	1263.901335
1034000	5000	0:00:03	1271.548293
1035000	5000	0:00:03	1266.36171
1036000	5000	0:00:03	1264.875249
1037000	5000	0:00:03	1259.672712
1038000	5000	0:00:03	1264.782781
1039000	5000	0:00:03	1261.776794
1040000	5000	0:00:03	1259.928552
1041000	5000	0:00:03	1263.122261
1042000	5000	0:00:03	1272.228698
1043000	5000	0:00:03	1262.802609
1044000	5000	0:00:03	1258.122438
1045000	5000	0:00:03	1259.090634
1046000	5000	0:00:04	1249.048537
1047000	5000	0:00:04	1244.757703
1048000	5000	0:00:04	1231.023162
1049000	5000	0:00:04	1219.460442
1050000	5000	0:00:04	1220.637861
1051000	5000	0:00:04	1220.096056
1052000	5000	0:00:04	1217.078039
1053000	5000	0:00:04	1205.392637

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
1054000	5000	0:00:04	1210.603336
1055000	5000	0:00:04	1205.187222
1056000	5000	0:00:04	1215.369564
1057000	5000	0:00:04	1212.903645
1058000	5000	0:00:04	1207.534046
1059000	5000	0:00:04	1212.032477
1060000	5000	0:00:04	1212.082425
1061000	5000	0:00:04	1231.332385
1062000	5000	0:00:04	1229.694062
1063000	5000	0:00:04	1223.724494
1064000	5000	0:00:04	1219.372413
1065000	5000	0:00:04	1217.23389
1066000	5000	0:00:04	1226.207588
1067000	5000	0:00:04	1220.580352
1068000	5000	0:00:04	1223.73857
1069000	5000	0:00:04	1228.830024
1070000	5000	0:00:04	1214.957881
1071000	5000	0:00:04	1223.864077
1072000	5000	0:00:04	1217.763665
1073000	5000	0:00:04	1215.846566
1074000	5000	0:00:04	1198.645531
1075000	5000	0:00:04	1200.944423
1076000	5000	0:00:04	1203.813295
1077000	5000	0:00:04	1200.789543
1078000	5000	0:00:04	1204.827406
1079000	5000	0:00:04	1195.222076
1080000	5000	0:00:04	1178.777108
1081000	5000	0:00:04	1186.271142
1082000	5000	0:00:04	1191.26583
1083000	5000	0:00:04	1196.825062
1084000	5000	0:00:04	1189.87049
1085000	5000	0:00:04	1180.269668
1086000	5000	0:00:04	1197.215946
1087000	5000	0:00:04	1210.383835

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
1088000	5000	0:00:04	1213.753969
1089000	5000	0:00:04	1201.158493
1090000	5000	0:00:04	1208.340936
1091000	5000	0:00:04	1206.404755
1092000	5000	0:00:04	1208.895439
1093000	5000	0:00:04	1215.706145
1094000	5000	0:00:04	1217.728372
1095000	5000	0:00:04	1204.223839
1096000	5000	0:00:04	1219.372116
1097000	5000	0:00:04	1212.602726
1098000	5000	0:00:04	1203.117518
1099000	5000	0:00:04	1201.527382
1100000	5000	0:00:04	1181.414181
1101000	5000	0:00:04	1188.353565
1102000	5000	0:00:04	1179.500656
1103000	5000	0:00:04	1187.10904
1104000	5000	0:00:04	1178.340129
1105000	5000	0:00:04	1181.408877
1106000	5000	0:00:04	1148.897323
1107000	5000	0:00:04	1167.548867
1108000	5000	0:00:04	1164.208867
1109000	5000	0:00:04	1178.281815
1110000	5000	0:00:04	1172.563606
1111000	5000	0:00:04	1176.922804
1112000	5000	0:00:04	1176.732792
1113000	5000	0:00:04	1180.944562
1114000	5000	0:00:04	1189.936469
1115000	5000	0:00:04	1166.44547
1116000	5000	0:00:04	1168.213381
1117000	5000	0:00:04	1173.930203
1118000	5000	0:00:04	1172.847455
1119000	5000	0:00:04	1160.50612
1120000	5000	0:00:04	1161.712736
1121000	5000	0:00:04	1157.322751

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
1122000	5000	0:00:04	1153.506047
1123000	5000	0:00:04	1155.235458
1124000	5000	0:00:04	1161.487131
1125000	5000	0:00:04	1155.355048
1126000	5000	0:00:04	1147.113392
1127000	5000	0:00:04	1144.661653
1128000	5000	0:00:04	1135.849922
1129000	5000	0:00:04	1136.27919
1130000	5000	0:00:04	1142.382173
1131000	5000	0:00:04	1142.481625
1132000	5000	0:00:04	1136.234261
1133000	5000	0:00:04	1135.228923
1134000	5000	0:00:04	1133.267755
1135000	5000	0:00:04	1137.521582
1136000	5000	0:00:04	1130.31491
1137000	5000	0:00:04	1132.415111
1138000	5000	0:00:04	1132.845379
1139000	5000	0:00:04	1135.97792
1140000	5000	0:00:04	1134.171592
1141000	5000	0:00:04	1142.486324
1142000	5000	0:00:04	1147.158923
1143000	5000	0:00:04	1137.480694
1144000	5000	0:00:04	1130.604747
1145000	5000	0:00:04	1143.354731
1146000	5000	0:00:04	1138.830239
1147000	5000	0:00:04	1140.278159
1148000	5000	0:00:04	1142.458653
1149000	5000	0:00:04	1142.949885
1150000	5000	0:00:04	1149.1075
1151000	5000	0:00:04	1136.769256
1152000	5000	0:00:04	1127.267612
1153000	5000	0:00:04	1124.004441
1154000	5000	0:00:04	1126.839283
1155000	5000	0:00:04	1110.877333

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
1156000	5000	0:00:04	1101.178724
1157000	5000	0:00:04	1126.133989
1158000	5000	0:00:04	1119.38987
1159000	5000	0:00:04	1114.476583
1160000	5000	0:00:04	1103.696478
1161000	5000	0:00:04	1115.10318
1162000	5000	0:00:04	1111.706986
1163000	5000	0:00:04	1127.924454
1164000	5000	0:00:04	1121.534403
1165000	5000	0:00:04	1121.410645
1166000	5000	0:00:04	1133.201489
1167000	5000	0:00:04	1124.952021
1168000	5000	0:00:04	1132.488723
1169000	5000	0:00:04	1122.868039
1170000	5000	0:00:04	1114.994264
1171000	5000	0:00:04	1117.044604
1172000	5000	0:00:04	1128.181755
1173000	5000	0:00:04	1122.805757
1174000	5000	0:00:04	1118.840559
1175000	5000	0:00:04	1119.06643
1176000	5000	0:00:04	1128.149427
1177000	5000	0:00:04	1119.065178
1178000	5000	0:00:04	1107.930612
1179000	5000	0:00:04	1104.282474
1180000	5000	0:00:04	1091.396608
1181000	5000	0:00:04	1095.957189
1182000	5000	0:00:04	1098.971868
1183000	5000	0:00:04	1094.521809
1184000	5000	0:00:04	1075.543139
1185000	5000	0:00:04	1086.558275
1186000	5000	0:00:04	1083.780573
1187000	5000	0:00:04	1090.009021
1188000	5000	0:00:04	1091.925018
1189000	5000	0:00:04	1091.759552

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
1190000	5000	0:00:04	1095.752796
1191000	5000	0:00:04	1097.252086
1192000	5000	0:00:04	1094.356513
1193000	5000	0:00:04	1101.417171
1194000	5000	0:00:04	1095.780413
1195000	5000	0:00:04	1091.079141
1196000	5000	0:00:04	1097.546654
1197000	5000	0:00:04	1096.328462
1198000	5000	0:00:04	1090.992482
1199000	5000	0:00:04	1093.709943
1200000	5000	0:00:04	1087.540957
1201000	5000	0:00:04	1088.334211
1202000	5000	0:00:04	1076.8969
1203000	5000	0:00:04	1072.620483
1204000	5000	0:00:04	1067.266613
1205000	5000	0:00:04	1063.112526
1206000	5000	0:00:04	1067.996793
1207000	5000	0:00:04	1075.349295
1208000	5000	0:00:04	1072.256813
1209000	5000	0:00:04	1072.174958
1210000	5000	0:00:04	1049.487315
1211000	5000	0:00:04	1072.847412
1212000	5000	0:00:04	1072.577685
1213000	5000	0:00:04	1074.325721
1214000	5000	0:00:04	1076.633479
1215000	5000	0:00:04	1085.636058
1216000	5000	0:00:04	1085.689805
1217000	5000	0:00:04	1073.957204
1218000	5000	0:00:04	1086.387113
1219000	5000	0:00:04	1082.898712
1220000	5000	0:00:04	1084.56929
1221000	5000	0:00:04	1087.467158
1222000	5000	0:00:04	1084.858264
1223000	5000	0:00:04	1076.879041

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
1224000	5000	0:00:04	1085.855559
1225000	5000	0:00:04	1083.678395
1226000	5000	0:00:04	1058.591337
1227000	5000	0:00:04	1072.908188
1228000	5000	0:00:04	1064.678132
1229000	5000	0:00:04	1059.815117
1230000	5000	0:00:04	1061.476697
1231000	5000	0:00:04	1054.878359
1232000	5000	0:00:04	1049.419472
1233000	5000	0:00:04	1060.662469
1234000	5000	0:00:04	1058.746677
1235000	5000	0:00:04	1058.314617
1236000	5000	0:00:04	1066.971678
1237000	5000	0:00:04	1064.723702
1238000	5000	0:00:04	1068.496845
1239000	5000	0:00:04	1065.781301
1240000	5000	0:00:04	1075.808341
1241000	5000	0:00:04	1070.911933
1242000	5000	0:00:04	1068.363056
1243000	5000	0:00:04	1079.033135
1244000	5000	0:00:04	1063.762793
1245000	5000	0:00:04	1071.891782
1246000	5000	0:00:04	1050.177701
1247000	5000	0:00:04	1054.492369
1248000	5000	0:00:04	1053.344749
1249000	5000	0:00:04	1048.601197
1250000	5000	0:00:04	1030.632883
1251000	5000	0:00:04	1021.058932
1252000	5000	0:00:04	1026.778591
1253000	5000	0:00:04	1023.293433
1254000	5000	0:00:04	1028.74414
1255000	5000	0:00:04	1021.459433
1256000	5000	0:00:04	1013.73263
1257000	5000	0:00:04	1015.585995

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
1258000	5000	0:00:04	1011.340566
1259000	5000	0:00:04	1012.596498
1260000	5000	0:00:04	1032.496804
1261000	5000	0:00:04	1023.673054
1262000	5000	0:00:04	1041.772797
1263000	5000	0:00:04	1038.408875
1264000	5000	0:00:04	1032.341824
1265000	5000	0:00:04	1038.806918
1266000	5000	0:00:04	1039.193818
1267000	5000	0:00:04	1034.797765
1268000	5000	0:00:05	997.303093
1269000	5000	0:00:04	1023.052022
1270000	5000	0:00:05	986.281806
1271000	5000	0:00:04	1023.754378
1272000	5000	0:00:04	1016.614945
1273000	5000	0:00:04	1004.441641
1274000	5000	0:00:04	1004.20884
1275000	5000	0:00:04	1008.934112
1276000	5000	0:00:04	1003.149488
1277000	5000	0:00:05	999.327453
1278000	5000	0:00:04	1009.189274
1279000	5000	0:00:05	993.960695
1280000	5000	0:00:05	991.216828
1281000	5000	0:00:05	998.70488
1282000	5000	0:00:05	993.104478
1283000	5000	0:00:04	1007.372354
1284000	5000	0:00:05	995.57923
1285000	5000	0:00:04	1010.100806
1286000	5000	0:00:04	1017.098649
1287000	5000	0:00:04	1022.783526
1288000	5000	0:00:04	1003.124734
1289000	5000	0:00:04	1015.231518
1290000	5000	0:00:05	961.883815
1291000	5000	0:00:05	991.641258

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
1292000	5000	0:00:05	998.773307
1293000	5000	0:00:04	1001.282844
1294000	5000	0:00:05	996.123089
1295000	5000	0:00:05	995.71385
1296000	5000	0:00:05	985.103851
1297000	5000	0:00:05	988.762125
1298000	5000	0:00:05	979.556845
1299000	5000	0:00:05	989.250801
1300000	5000	0:00:05	991.844656
1301000	5000	0:00:05	982.018071
1302000	5000	0:00:05	986.6917
1303000	5000	0:00:05	977.840377
1304000	5000	0:00:05	966.769428
1305000	5000	0:00:05	980.729451
1306000	5000	0:00:05	972.118287
1307000	5000	0:00:05	982.614407
1308000	5000	0:00:04	1002.914067
1309000	5000	0:00:05	999.855021
1310000	5000	0:00:05	996.736088
1311000	5000	0:00:05	999.507443
1312000	5000	0:00:05	987.415781
1313000	5000	0:00:05	990.078424
1314000	5000	0:00:05	996.681449
1315000	5000	0:00:05	983.083686
1316000	5000	0:00:05	981.656569
1317000	5000	0:00:05	971.432124
1318000	5000	0:00:05	974.959149
1319000	5000	0:00:05	971.851677
1320000	5000	0:00:07	668.67805
1321000	5000	0:00:05	895.401843
1322000	5000	0:00:05	953.11585
1323000	5000	0:00:05	936.18951
1324000	5000	0:00:05	922.185269
1325000	5000	0:00:03	1516.186197

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
1326000	5000	0:00:02	1928.092187
1327000	5000	0:00:02	1901.224541
1328000	5000	0:00:02	1906.755099
1329000	5000	0:00:02	1913.127922
1330000	5000	0:00:02	1916.952996
1331000	5000	0:00:02	1883.545647
1332000	5000	0:00:02	1870.468556
1333000	5000	0:00:02	1906.627858
1334000	5000	0:00:02	1868.525326
1335000	5000	0:00:02	1880.638153
1336000	5000	0:00:02	1890.845999
1337000	5000	0:00:02	1870.530134
1338000	5000	0:00:02	1875.092583
1339000	5000	0:00:02	1876.874059
1340000	5000	0:00:02	1892.33521
1341000	5000	0:00:02	1849.47425
1342000	5000	0:00:02	1865.947356
1343000	5000	0:00:02	1867.186292
1344000	5000	0:00:02	1871.167148
1345000	5000	0:00:02	1879.782637
1346000	5000	0:00:02	1905.702051
1347000	5000	0:00:02	1889.685454
1348000	5000	0:00:02	1856.668876
1349000	5000	0:00:02	1857.546263
1350000	5000	0:00:02	1885.087338
1351000	5000	0:00:02	1872.547665
1352000	5000	0:00:02	1864.253974
1353000	5000	0:00:02	1900.227419
1354000	5000	0:00:02	1842.090817
1355000	5000	0:00:02	1882.799497
1356000	5000	0:00:02	1922.824737
1357000	5000	0:00:02	1855.49343
1358000	5000	0:00:02	1877.26445
1359000	5000	0:00:02	1882.040478

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
1360000	5000	0:00:02	1856.838495
1361000	5000	0:00:02	1893.897068
1362000	5000	0:00:02	1867.850336
1363000	5000	0:00:02	1863.68487
1364000	5000	0:00:02	1868.600743
1365000	5000	0:00:02	1864.637047
1366000	5000	0:00:02	1861.099448
1367000	5000	0:00:02	1874.627886
1368000	5000	0:00:02	1887.568136
1369000	5000	0:00:02	1854.029994
1370000	5000	0:00:02	1882.27641
1371000	5000	0:00:02	1853.369556
1372000	5000	0:00:02	1869.967683
1373000	5000	0:00:02	1855.515464
1374000	5000	0:00:02	1887.311642
1375000	5000	0:00:02	1880.136769
1376000	5000	0:00:02	1864.060759
1377000	5000	0:00:02	1836.649833
1378000	5000	0:00:02	1814.222854
1379000	5000	0:00:02	1813.268191
1380000	5000	0:00:02	1823.642134
1381000	5000	0:00:02	1830.377457
1382000	5000	0:00:02	1837.213344
1383000	5000	0:00:02	1820.119749
1384000	5000	0:00:02	1816.342359
1385000	5000	0:00:02	1840.816498
1386000	5000	0:00:02	1845.373225
1387000	5000	0:00:02	1804.201552
1388000	5000	0:00:02	1797.982951
1389000	5000	0:00:02	1812.664069
1390000	5000	0:00:02	1820.771949
1391000	5000	0:00:02	1805.758202
1392000	5000	0:00:02	1822.181341
1393000	5000	0:00:02	1825.569733

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
1394000	5000	0:00:02	1792.308702
1395000	5000	0:00:02	1832.606766
1396000	5000	0:00:02	1816.916584
1397000	5000	0:00:02	1863.848825
1398000	5000	0:00:02	1795.201283
1399000	5000	0:00:02	1814.359787
1400000	5000	0:00:02	1807.811554
1401000	5000	0:00:02	1818.51378
1402000	5000	0:00:02	1778.570151
1403000	5000	0:00:02	1826.327905
1404000	5000	0:00:02	1832.420055
1405000	5000	0:00:02	1849.274511
1406000	5000	0:00:02	1842.17769
1407000	5000	0:00:02	1830.430393
1408000	5000	0:00:02	1820.999401
1409000	5000	0:00:02	1819.148944
1410000	5000	0:00:02	1805.378729
1411000	5000	0:00:02	1798.768851
1412000	5000	0:00:02	1822.651623
1413000	5000	0:00:02	1797.014943
1414000	5000	0:00:02	1760.001295
1415000	5000	0:00:02	1797.55633
1416000	5000	0:00:02	1847.361395
1417000	5000	0:00:02	1777.712042
1418000	5000	0:00:02	1842.464156
1419000	5000	0:00:02	1796.636553
1420000	5000	0:00:02	1793.813997
1421000	5000	0:00:02	1792.703267
1422000	5000	0:00:02	1783.679617
1423000	5000	0:00:02	1784.065936
1424000	5000	0:00:02	1766.511673
1425000	5000	0:00:02	1770.608466
1426000	5000	0:00:02	1770.125799
1427000	5000	0:00:02	1784.357537

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
1428000	5000	0:00:02	1765.294245
1429000	5000	0:00:02	1790.782841
1430000	5000	0:00:02	1800.886757
1431000	5000	0:00:02	1813.917463
1432000	5000	0:00:02	1777.640623
1433000	5000	0:00:02	1810.136912
1434000	5000	0:00:02	1798.077352
1435000	5000	0:00:02	1759.994481
1436000	5000	0:00:02	1769.523775
1437000	5000	0:00:02	1800.314191
1438000	5000	0:00:02	1813.622043
1439000	5000	0:00:02	1786.63951
1440000	5000	0:00:02	1788.801246
1441000	5000	0:00:02	1807.667765
1442000	5000	0:00:02	1744.338575
1443000	5000	0:00:02	1772.2358
1444000	5000	0:00:02	1759.851384
1445000	5000	0:00:02	1745.601608
1446000	5000	0:00:02	1775.464151
1447000	5000	0:00:02	1752.307789
1448000	5000	0:00:02	1768.730055
1449000	5000	0:00:02	1727.829528
1450000	5000	0:00:02	1723.634511
1451000	5000	0:00:02	1776.419804
1452000	5000	0:00:02	1740.531941
1453000	5000	0:00:02	1777.805591
1454000	5000	0:00:02	1779.673565
1455000	5000	0:00:02	1721.563303
1456000	5000	0:00:02	1723.824076
1457000	5000	0:00:02	1739.488618
1458000	5000	0:00:02	1751.359493
1459000	5000	0:00:02	1735.168818
1460000	5000	0:00:02	1731.276162
1461000	5000	0:00:05	933.716755

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
1462000	5000	0:00:03	1539.456264
1463000	5000	0:00:02	1791.953484
1464000	5000	0:00:02	1754.718438
1465000	5000	0:00:02	1733.407477
1466000	5000	0:00:02	1755.080607
1467000	5000	0:00:02	1748.043328
1468000	5000	0:00:02	1778.904259
1469000	5000	0:00:02	1788.677102
1470000	5000	0:00:02	1768.700649
1471000	5000	0:00:02	1754.211775
1472000	5000	0:00:02	1735.769382
1473000	5000	0:00:02	1772.177382
1474000	5000	0:00:02	1760.884556
1475000	5000	0:00:02	1762.471069
1476000	5000	0:00:02	1718.501838
1477000	5000	0:00:02	1729.785382
1478000	5000	0:00:02	1689.8777
1479000	5000	0:00:02	1757.532874
1480000	5000	0:00:02	1731.622721
1481000	5000	0:00:02	1711.154985
1482000	5000	0:00:03	1638.322594
1483000	5000	0:00:02	1739.782778
1484000	5000	0:00:02	1717.649953
1485000	5000	0:00:02	1737.559681
1486000	5000	0:00:02	1728.839181
1487000	5000	0:00:02	1736.759381
1488000	5000	0:00:02	1725.581469
1489000	5000	0:00:02	1736.706295
1490000	5000	0:00:02	1718.575672
1491000	5000	0:00:02	1740.628889
1492000	5000	0:00:02	1743.5077
1493000	5000	0:00:02	1734.235367
1494000	5000	0:00:02	1718.513651
1495000	5000	0:00:02	1721.209501

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
1496000	5000	0:00:02	1726.062789
1497000	5000	0:00:04	1205.88685
1498000	5000	0:00:06	820.790559
1499000	5000	0:00:06	717.949719
1500000	5000	0:00:04	1118.487662
1501000	5000	0:00:04	1018.050649
1502000	5000	0:00:06	832.176192
1503000	5000	0:00:04	1044.53413
1504000	5000	0:00:04	1139.705044
1505000	5000	0:00:03	1331.157335
1506000	5000	0:00:03	1548.225362
1507000	5000	0:00:02	1671.175498
1508000	5000	0:00:04	1009.730368
1509000	5000	0:00:05	932.743243
1510000	5000	0:00:05	913.580743
1511000	5000	0:00:05	900.155961
1512000	5000	0:00:05	887.932448
1513000	5000	0:00:06	828.200286
1514000	5000	0:00:05	876.94513
1515000	5000	0:00:05	902.483092
1516000	5000	0:00:05	946.031012
1517000	5000	0:00:06	807.948664
1518000	5000	0:00:05	899.080708
1519000	5000	0:00:05	916.545759
1520000	5000	0:00:05	913.825522
1521000	5000	0:00:06	833.064115
1522000	5000	0:00:06	789.880057
1523000	5000	0:00:07	688.724492
1524000	5000	0:00:06	763.372414
1525000	5000	0:00:04	1024.987557
1526000	5000	0:00:07	646.395698
1527000	5000	0:00:05	900.393598
1528000	5000	0:00:05	952.715764
1529000	5000	0:00:06	828.857091

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
1530000	5000	0:00:05	956.115997
1531000	5000	0:00:05	975.285486
1532000	5000	0:00:05	936.762709
1533000	5000	0:00:05	842.117461
1534000	5000	0:00:08	620.019077
1535000	5000	0:00:06	775.29621
1536000	5000	0:00:06	781.772322
1537000	5000	0:00:05	970.701699
1538000	5000	0:00:03	1253.843972
1539000	5000	0:00:04	1059.298693
1540000	5000	0:00:04	1023.641198
1541000	5000	0:00:04	1056.212027
1542000	5000	0:00:03	1582.460764
1543000	5000	0:00:03	1574.771902
1544000	5000	0:00:03	1389.042069
1545000	5000	0:00:03	1330.815076
1546000	5000	0:00:05	931.180023
1547000	5000	0:00:06	831.712603
1548000	5000	0:00:06	803.483648
1549000	5000	0:00:03	1497.619384
1550000	5000	0:00:03	1472.06463
1551000	5000	0:00:03	1330.719445
1552000	5000	0:00:03	1592.257299
1553000	5000	0:00:03	1607.98797
1554000	5000	0:00:07	686.018329
1555000	5000	0:00:04	1196.533499
1556000	5000	0:00:03	1258.088882
1557000	5000	0:00:05	856.625993
1558000	5000	0:00:04	1066.71218
1559000	5000	0:00:03	1566.256568
1560000	5000	0:00:05	932.937469
1561000	5000	0:00:04	1249.150266
1562000	5000	0:00:05	864.098405
1563000	5000	0:00:04	1116.875429

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
1564000	5000	0:00:03	1558.12799
1565000	5000	0:00:05	859.083945
1566000	5000	0:00:07	706.44109
1567000	5000	0:00:04	1023.132829
1568000	5000	0:00:04	1237.650415
1569000	5000	0:00:04	1231.512534
1570000	5000	0:00:03	1557.96826
1571000	5000	0:00:03	1598.007221
1572000	5000	0:00:03	1606.432929
1573000	5000	0:00:03	1265.912841
1574000	5000	0:00:03	1381.897638
1575000	5000	0:00:03	1589.741715
1576000	5000	0:00:05	954.768237
1577000	5000	0:00:05	848.91128
1578000	5000	0:00:04	1164.456412
1579000	5000	0:00:05	953.224874
1580000	5000	0:00:03	1530.759229
1581000	5000	0:00:03	1545.134141
1582000	5000	0:00:03	1629.795265
1583000	5000	0:00:03	1594.86123
1584000	5000	0:00:03	1561.119385
1585000	5000	0:00:03	1596.561135
1586000	5000	0:00:03	1592.774664
1587000	5000	0:00:03	1594.330306
1588000	5000	0:00:03	1597.455828
1589000	5000	0:00:03	1595.938401
1590000	5000	0:00:03	1612.656645
1591000	5000	0:00:03	1599.722544
1592000	5000	0:00:03	1620.967802
1593000	5000	0:00:03	1585.355375
1594000	5000	0:00:03	1602.73876
1595000	5000	0:00:03	1593.833268
1596000	5000	0:00:03	1625.236147
1597000	5000	0:00:03	1585.304104

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
1598000	5000	0:00:03	1590.51392
1599000	5000	0:00:03	1585.925608
1600000	5000	0:00:03	1611.576534
1601000	5000	0:00:03	1599.310633
1602000	5000	0:00:03	1618.138687
1603000	5000	0:00:03	1619.254884
1604000	5000	0:00:03	1573.884596
1605000	5000	0:00:03	1588.0341
1606000	5000	0:00:04	1167.462721
1607000	5000	0:00:03	1474.46782
1608000	5000	0:00:03	1613.791332
1609000	5000	0:00:03	1578.12143
1610000	5000	0:00:03	1585.802878
1611000	5000	0:00:04	1077.13098
1612000	5000	0:00:03	1460.81282
1613000	5000	0:00:03	1522.059822
1614000	5000	0:00:03	1381.159765
1615000	5000	0:00:03	1496.494312
1616000	5000	0:00:03	1536.505685
1617000	5000	0:00:03	1520.022034
1618000	5000	0:00:03	1561.397263
1619000	5000	0:00:03	1535.979396
1620000	5000	0:00:03	1528.735488
1621000	5000	0:00:03	1360.477582
1622000	5000	0:00:03	1498.407043
1623000	5000	0:00:03	1520.285935
1624000	5000	0:00:03	1508.241939
1625000	5000	0:00:03	1444.107133
1626000	5000	0:00:05	914.226535
1627000	5000	0:00:03	1555.737888
1628000	5000	0:00:03	1563.245961
1629000	5000	0:00:03	1309.351256
1630000	5000	0:00:03	1565.594005
1631000	5000	0:00:03	1612.402341

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
1632000	5000	0:00:03	1591.53811
1633000	5000	0:00:03	1411.698007
1634000	5000	0:00:06	782.236596
1635000	5000	0:00:05	895.865652
1636000	5000	0:00:03	1552.618071
1637000	5000	0:00:04	1046.081348
1638000	5000	0:00:06	806.387102
1639000	5000	0:00:05	938.367814
1640000	5000	0:00:03	1509.440797
1641000	5000	0:00:07	653.841522
1642000	5000	0:00:06	761.758272
1643000	5000	0:00:06	779.165367
1644000	5000	0:00:07	707.64417
1645000	5000	0:00:07	704.180622
1646000	5000	0:00:08	598.11925
1647000	5000	0:00:04	1043.461856
1648000	5000	0:00:05	915.991993
1649000	5000	0:00:03	1275.319557
1650000	5000	0:00:03	1502.207945
1651000	5000	0:00:03	1396.140676
1652000	5000	0:00:06	807.591098
1653000	5000	0:00:04	1148.888347
1654000	5000	0:00:04	1209.548466
1655000	5000	0:00:03	1466.106262
1656000	5000	0:00:07	651.721222
1657000	5000	0:00:07	638.200784
1658000	5000	0:00:03	1414.692487
1659000	5000	0:00:03	1268.05038
1660000	5000	0:00:03	1553.671746
1661000	5000	0:00:03	1511.01591
1662000	5000	0:00:06	756.686078
1663000	5000	0:00:06	772.802544
1664000	5000	0:00:05	954.712816
1665000	5000	0:00:07	625.940397

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)
1666000	5000	0:00:06	740.73701
1667000	5000	0:00:04	1204.862826
1668000	5000	0:00:06	784.182475
1669000	5000	0:00:07	710.875037
1670000	5000	0:00:05	902.490911
1671000	5000	0:00:03	1419.627371
1672000	5000	0:00:03	1487.436959
1673000	5000	0:00:04	1145.464613
1674000	5000	0:00:03	1369.153305
1675000	5000	0:00:06	808.58785
1676000	5000	0:00:03	1528.742499
1677000	5000	0:00:04	1121.851971
1678000	5000	0:00:03	1494.726753
1679000	5000	0:00:03	1489.958129
1680000	5000	0:00:03	1465.726335
1681000	5000	0:00:03	1520.952798
1682000	5000	0:00:04	1230.005946
1683000	5000	0:00:04	1038.777135
1684000	5000	0:00:04	1107.358642
1685000	5000	0:00:03	1474.064861
1686000	5000	0:00:03	1468.016616
1687000	5000	0:00:03	1469.540107
1688000	5000	0:00:06	719.018189
1689000	5000	0:00:03	1499.285441
1690000	5000	0:00:05	838.554875
1691000	5000	0:00:06	826.98553
1692000	5000	0:00:06	818.555809
1693000	5000	0:00:05	834.537014
1694000	5000	0:00:05	888.657285
1695000	5000	0:00:05	967.084879
1696000	5000	0:00:04	1121.593524
1697000	5000	0:00:03	1433.042932
1698000	5000	0:00:03	1436.769223
1699000	5000	0:00:05	851.616982

Payload (bytes)	Total Messages	Final Rate (bytes/sec)	Avg. Rate (bytes/sec)

B.3 Broadcast Policy Throughput

Total Messages	Payload Size (kiB)	Input Rate (bytes/sec)	Output A Rate (bytes/sec)	Output B Rate (bytes/sec)	Delta Rate Input / Output (bytes/sec)
1000	125	194.789118	186.214552	186.217685	
2000	250	319.579995	308.383092	308.439943	
3000	375	405.679513	393.631629	393.699184	
4000	500	469.921665	457.83648	457.872572	
5000	625	516.91175	505.204026	505.277142	
6000	750	557.304201	545.972406	545.99801	
7000	875	588.297881	577.487688	577.534169	
8000	1000	614.60997	604.276047	604.320301	
9000	1125	636.717639	626.76063	626.776267	
10000	1250	656.952917	647.489717	647.516435	
11000	1375	671.040468	662.060541	662.10503	
12000	1500	684.757816	676.197313	676.238253	
13000	1625	696.102522	687.936554	687.992165	
14000	1750	706.178248	698.430692	698.478827	
15000	1875	716.737092	709.286881	709.346468	
16000	2000	727.907264	720.669872	720.700328	
17000	2125	732.653839	725.716549	725.739574	
18000	2250	741.069953	734.365485	734.40945	
19000	2375	746.180469	739.717441	739.74766	
20000	2500	753.378271	747.13386	747.175653	
21000	2625	757.258458	751.301117	751.297353	
22000	2750	762.787667	756.981195	757.021212	
23000	2875	766.245155	760.659294	760.670516	
24000	3000	771.731476	766.21406	766.262383	
25000	3125	775.789147	770.49596	770.53458	
26000	3250	779.414605	774.177217	774.306718	
27000	3375	781.176697	776.203949	776.229169	
28000	3500	785.952447	781.228308	781.250654	
29000	3625	787.994764	783.373917	783.370235	
30000	3750	791.44033	786.871842	786.891597	
31000	3875	793.489171	788.995736	789.021123	
32000	4000	794.327193	789.871417	789.999025	
33000	4125	800.858209	796.617471	796.630453	

Total Messages	Payload Size (kiB)	Input Rate (bytes/sec)	Output A Rate (bytes/sec)	Output B Rate (bytes/sec)	Delta Rate Input / Output (bytes/sec)
34000	4250	802.433393	798.091268	798.211548	
35000	4375	803.334674	799.255901	799.267803	
36000	4500	803.653695	799.624018	799.733652	
37000	4625	808.208815	804.202688	804.310132	
38000	4750	811.207801	807.45226	807.472354	
39000	4875	811.567509	807.795725	807.80364	
40000	5000	812.413945	808.751748	808.772486	
41000	5125	814.721355	811.137814	811.151921	
42000	5250	816.953354	813.41747	813.430059	
43000	5375	817.040578	813.592063	813.59739	
44000	5500	819.830431	816.468948	816.488631	
45000	5625	821.882981	818.493552	818.490931	
46000	5750	823.004346	819.760002	819.772157	
47000	5875	824.273905	821.060306	821.075469	
48000	6000	826.321436	822.971452	822.981784	
49000	6125	825.761633	822.422164	822.435366	
50000	6250	826.040347	822.878606	822.896135	
51000	6375	828.678099	825.430881	825.440865	
52000	6500	829.015055	824.780884	824.804001	
53000	6625	830.03543	827.101547	827.102529	
54000	6750	830.895578	828.062124	828.072335	
55000	6875	833.946069	831.181751	831.19008	
56000	7000	832.460427	829.711206	829.718645	
57000	7125	833.832694	831.138027	831.150329	
58000	7250	834.314072	831.650687	831.656542	
59000	7375	833.149534	830.490382	830.496895	
60000	7500	836.641063	834.076	834.094333	
61000	7625	837.621256	835.092997	835.109198	
62000	7750	836.521165	833.928026	833.940984	
63000	7875	840.221617	837.783454	837.786484	
64000	8000	838.402118	835.960335	835.976093	
65000	8125	839.971499	837.534537	837.529756	
66000	8250	841.124247	838.722834	838.742341	
67000	8375	841.6565	839.304061	839.310696	

Total Messages	Payload Size (kiB)	Input Rate (bytes/sec)	Output A Rate (bytes/sec)	Output B Rate (bytes/sec)	Delta Rate Input / Output (bytes/sec)
68000	8500	841.393054	838.957357	838.960888	
69000	8625	842.506952	840.250149	840.261364	
70000	8750	846.4707	844.073967	844.080014	
71000	8875	843.948783	841.743965	841.743486	
72000	9000	843.926793	841.697489	841.690916	
73000	9125	846.436214	844.12986	844.147124	
74000	9250	844.282624	839.25538	839.259621	
75000	9375	846.505371	844.119796	844.129786	
76000	9500	847.011548	844.926816	844.928611	
77000	9625	846.648857	842.857863	842.863189	
78000	9750	847.71148	845.67537	845.689738	
79000	9875	846.58075	844.516914	844.572633	
80000	10000	845.986525	843.96287	843.971044	
81000	10125	845.766157	843.775926	843.779152	
82000	10250	845.880265	843.978984	843.983292	
83000	10375	846.122802	844.154199	844.160561	
84000	10500	847.870887	845.999377	846.006143	
85000	10625	850.18767	848.334193	848.32184	
86000	10750	851.660253	849.822619	849.826113	
87000	10875	851.036532	849.120151	849.167877	
88000	11000	849.560272	847.759831	847.771397	
89000	11125	850.787835	848.921539	848.925677	
90000	11250	851.557733	849.482513	849.490759	
91000	11375	808.860388	789.336934	789.349798	
92000	11500	826.785841	816.218332	816.253129	
93000	11625	844.575941	840.758985	840.767116	
94000	11750	860.879351	858.955087	858.964108	
95000	11875	838.025563	832.419269	832.426759	
96000	12000	849.247802	844.85976	844.865944	
97000	12125	835.553112	829.742915	829.746682	
98000	12250	822.236538	814.017729	814.01976	
99000	12375	839.918549	834.822514	834.829301	
100000	12500	828.725983	819.504766	819.507695	
101000	12625	834.679861	825.856921	825.86494	

Total Messages	Payload Size (kiB)	Input Rate (bytes/sec)	Output A Rate (bytes/sec)	Output B Rate (bytes/sec)	Delta Rate Input / Output (bytes/sec)
102000	12750	832.684377	822.039951	822.045444	
103000	12875	851.30211	847.386785	847.394427	
104000	13000	853.933304	850.782522	850.790326	
105000	13125	831.966122	823.806903	823.809437	
106000	13250	831.44044	821.574307	821.576997	
107000	13375	836.929156	831.461408	831.470336	
108000	13500	860.299124	858.73307	858.740705	
109000	13625	862.715129	861.207379	861.21581	
110000	13750	863.485698	861.974947	861.979588	
111000	13875	860.808419	859.339791	859.345653	
112000	14000	862.594071	861.123788	861.130052	
113000	14125	861.463687	860.058748	860.067946	
114000	14250	863.627682	862.142138	862.182397	
115000	14375	859.409827	857.723304	857.73092	
116000	14500	815.549899	799.210436	799.244664	
117000	14625	828.267516	819.699443	819.697763	
118000	14750	854.317841	851.519491	851.517755	

B.4 RoundRobin Policy Throughput

Total Messages	Payload Size (kiB)	Input Rate (bytes/sec)	Output A Rate (bytes/sec)	Output B Rate (bytes/sec)	Delta Rate Input / Output (bytes/sec)
1000	125	245.959682	115.49566	115.271805	15.192217
2000	250	483.975452	225.119908	225.095841	33.759703
3000	375	706.946765	330.694255	330.893226	45.359284
4000	500	925.602179	431.859617	432.422002	61.32056
5000	625	1121.73972	530.010942	530.057371	61.671407
6000	750	1327.3608	625.673961	625.785885	75.900954
7000	875	1528.43901	714.76369	714.845177	98.830143
8000	1000	1717.008276	784.931738	784.802502	147.274036
9000	1125	1896.617888	894.605529	894.75359	107.258769
10000	1250	2093.970705	966.555601	966.521734	160.89337
11000	1375	2228.154966	1052.897816	1053.015719	122.241431
12000	1500	2398.183616	1137.725721	1138.023498	122.434397
13000	1625	2530.591937	1199.568732	1199.555069	131.468136
14000	1750	2704.231272	1275.288854	1275.456952	153.485466
15000	1875	2835.186632	1350.283734	1350.234954	134.667944
16000	2000	2990.326667	1417.825095	1418.01189	154.489682
17000	2125	3097.797466	1472.829571	1472.889753	152.078142
18000	2250	3238.233608	1509.825369	1509.885841	218.522398
19000	2375	3410.723242	1619.777663	1620.170647	170.774932
20000	2500	3500.121717	1673.046394	1673.242271	153.833052
21000	2625	3627.407238	1735.219296	1735.366744	156.821198
22000	2750	3738.728795	1779.030671	1779.597584	180.10054
23000	2875	3829.203537	1822.000871	1822.285815	184.916851
24000	3000	3969.099896	1895.407753	1895.589286	178.102857
25000	3125	4047.62271	1927.038977	1927.348656	193.235077
26000	3250	4176.439133	1970.585991	1970.945181	234.907961
27000	3375	4301.003328	2054.869601	2055.013833	191.119894
28000	3500	4350.946067	2087.176415	2087.425154	176.344498
29000	3625	4557.236215	2160.96618	2161.293017	234.977018
30000	3750	4598.552498	2188.677625	2188.897005	220.977868
31000	3875	4647.510329	2233.326445	2233.91146	180.272424
32000	4000	4719.221093	2256.542392	2256.941822	205.736879
33000	4125	4822.876922	2318.246975	2318.507644	186.122303

Total Messages	Payload Size (kiB)	Input Rate (bytes/sec)	Output A Rate (bytes/sec)	Output B Rate (bytes/sec)	Delta Rate Input / Output (bytes/sec)
34000	4250	4916.091713	2366.851379	2367.332758	181.907576
35000	4375	5056.513034	2411.117703	2411.169	234.226331
36000	4500	5124.180238	2444.469435	2444.727732	234.983071
37000	4625	5254.207058	2522.582616	2522.709374	208.915068
38000	4750	5163.550713	2488.687427	2489.26773	185.595556
39000	4875	5366.6266	2563.146569	2563.34253	240.137501
40000	5000	5362.090583	2561.581867	2562.10074	238.407976
41000	5125	5457.820631	2628.769336	2628.992582	200.058713
42000	5250	5512.943999	2644.795297	2644.809261	223.339441
43000	5375	5629.895636	2711.541999	2711.819039	206.534598
44000	5500	5678.591709	2732.572193	2733.022248	212.997268
45000	5625	5688.518711	2701.495343	2701.478536	285.544832
46000	5750	5668.874858	2735.718719	2736.052187	197.103952
47000	5875	5843.543475	2811.338963	2811.840395	220.364117
48000	6000	5878.557813	2836.998884	2837.465043	204.093886
49000	6125	5853.435704	2821.295498	2821.35263	210.787576
50000	6250	6081.740292	2914.743489	2916.814	250.182803
51000	6375	6029.889809	2890.29519	2890.641186	248.953433
52000	6500	6167.700488	2977.356592	2977.728658	212.615238
53000	6625	6260.699299	3020.817724	3021.234936	218.646639
54000	6750	6305.28509	3043.679311	3044.065245	217.540534
55000	6875	6285.168374	3043.950539	3044.090968	197.126867
56000	7000	6345.458787	3061.1986	3061.531199	222.728988
57000	7125	6546.293724	3140.466644	3140.915488	264.911592
58000	7250	6491.900235	3131.983057	3132.36602	227.551158
59000	7375	6534.090343	3161.471949	3162.078567	210.539827
60000	7500	6594.080494	3194.123426	3194.603599	205.353469
61000	7625	6693.615969	3224.601024	3224.870185	244.14476
62000	7750	6622.084374	3181.249374	3181.451885	259.383115
63000	7875	6725.969791	3268.382914	3268.77849	188.808387
64000	8000	6774.89921	3278.189744	3278.601404	218.108062
65000	8125	6826.548637	3294.490985	3294.558599	237.499053
66000	8250	6886.014754	3321.867369	3322.381373	241.766012
67000	8375	6838.609031	3321.986455	3322.831352	193.791224

Total Messages	Payload Size (kiB)	Input Rate (bytes/sec)	Output A Rate (bytes/sec)	Output B Rate (bytes/sec)	Delta Rate Input / Output (bytes/sec)
68000	8500	6997.678829	3397.055998	3397.494794	203.128037
69000	8625	6990.329032	3394.683325	3395.026232	200.619475
70000	8750	6859.756396	3332.863108	3332.995615	193.897673
71000	8875	7135.378634	3464.61647	3464.76033	206.001834
72000	9000	7001.864052	3388.62565	3389.033986	224.204416
73000	9125	7176.097501	3497.260339	3497.477023	181.360139
74000	9250	7073.522962	3444.162791	3444.370345	184.989826
75000	9375	7100.814236	3445.47898	3445.896908	209.438348
76000	9500	7228.237987	3522.782696	3523.170286	182.285005
77000	9625	7262.871034	3538.166262	3538.411608	186.293164
78000	9750	7307.921131	3545.124421	3545.355006	217.441704
79000	9875	7229.050828	3523.318393	3523.661973	182.070462
80000	10000	7330.617156	3551.234733	3551.882337	227.500086
81000	10125	7286.84232	3546.189345	3546.520942	194.132033
82000	10250	7295.035524	3558.011487	3558.404232	178.619805
83000	10375	7516.827732	3645.556796	3645.91608	225.354856
84000	10500	7632.506445	3709.649262	3709.906634	212.950549
85000	10625	7461.986883	3613.203597	3613.517048	235.266238
86000	10750	7702.097209	3750.540389	3750.755535	200.801285
87000	10875	7580.866937	3694.292808	3694.46865	192.105479
88000	11000	7553.457449	3691.583068	3691.769417	170.104964
89000	11125	7526.924188	3665.755734	3666.061541	195.106913
90000	11250	7655.84428	3738.777555	3739.051359	178.015366
91000	11375	7636.644109	3716.994247	3717.258567	202.391295
92000	11500	7919.279814	3844.370176	3844.63825	230.271388
93000	11625	7874.280424	3841.699172	3841.759117	190.822135
94000	11750	7827.79582	3802.844922	3803.273959	221.676939
95000	11875	7885.966599	3854.582464	3854.777774	176.606361
96000	12000	7902.835949	3827.329942	3827.569603	247.936404
97000	12125	7807.157731	3795.809969	3795.974353	215.373409
98000	12250	7878.95798	3840.163705	3840.277459	198.516816
99000	12375	7814.692949	3800.132304	3800.515955	214.04469
100000	12500	7980.477836	3894.425218	3894.737334	191.315284
101000	12625	8149.846137	3957.542168	3957.780533	234.523436

Total Messages	Payload Size (kiB)	Input Rate (bytes/sec)	Output A Rate (bytes/sec)	Output B Rate (bytes/sec)	Delta Rate Input / Output (bytes/sec)
102000	12750	8009.171915	3910.295152	3910.562486	188.314277
103000	12875	8089.924776	3951.349134	3951.625227	186.950415
104000	13000	8090.922677	3957.235329	3957.692413	175.994935
105000	13125	7988.76217	3908.188137	3908.470457	172.103576
106000	13250	8151.410763	3969.845225	3970.095026	211.470512
107000	13375	8234.758021	4022.792183	4023.233838	188.732
108000	13500	8130.297962	3976.724977	3976.683924	176.889061
109000	13625	8206.0517	4007.510003	4007.748761	190.792936
110000	13750	8105.502398	3958.619671	3959.060237	187.82249
111000	13875	8107.096646	3964.320827	3964.714817	178.061002
112000	14000	8190.760471	4006.837511	4006.925051	176.997909
113000	14125	8222.493395	4028.041276	4028.294558	166.157561
114000	14250	8240.946778	4022.410005	4022.650648	195.886125
115000	14375	8309.227542	4072.468281	4072.785585	163.973676
116000	14500	8260.427936	4038.304578	4038.468227	183.655131
117000	14625	8416.182377	4100.612774	4102.427997	213.141606
118000	14750	8429.241553	4121.4346	4121.57784	186.229113
119000	14875	8218.666816	4007.226452	4007.678803	203.761561
120000	15000	8452.090487	4139.376622	4139.841745	172.87212
121000	15125	8386.130145	4113.039101	4113.3551	159.735944
122000	15250	8461.643992	4138.883377	4139.177859	183.582756
123000	15375	8354.361306	4102.296472	4102.428161	149.636673
124000	15500	8402.703638	4108.194363	4108.192067	186.317208
125000	15625	8347.456116	4068.95249	4069.37311	209.130516
126000	15750	8503.345121	4169.822485	4170.066832	163.455804
127000	15875	8404.71384	4123.161633	4123.520981	158.031226
128000	16000	8447.937637	4131.247643	4131.0987	185.591294
129000	16125	8525.782495	4153.799397	4154.010186	217.972912
130000	16250	8477.507477	4147.373177	4147.864994	182.269306
131000	16375	8603.40528	4197.968809	4197.802727	207.633744
132000	16500	8658.974174	4234.432692	4234.622917	189.918565
133000	16625	8600.596881	4201.444265	4201.586266	197.56635
134000	16750	8525.594503	4177.452065	4177.660565	170.481873
135000	16875	8672.958352	4250.677599	4250.968304	171.312449

Total Messages	Payload Size (kiB)	Input Rate (bytes/sec)	Output A Rate (bytes/sec)	Output B Rate (bytes/sec)	Delta Rate Input / Output (bytes/sec)
136000	17000	8677.678172	4254.104209	4254.609928	168.964035
137000	17125	8614.405562	4196.484629	4196.730414	221.190519
138000	17250	8763.013074	4306.809347	4307.035627	149.1681
139000	17375	8773.302349	4293.788008	4294.041899	185.472442
140000	17500	8685.830711	4250.890021	4251.107983	183.832707
141000	17625	8905.656507	4362.765122	4363.003313	179.888072
142000	17750	8627.013793	4234.644478	4234.812927	157.556388
143000	17875	8505.337932	4144.262198	4144.359445	216.716289
144000	18000	8745.129221	4288.650454	4288.782686	167.696081
145000	18125	8872.825607	4358.256103	4358.50925	156.060254
146000	18250	8639.94639	4242.276731	4242.591895	155.077764
147000	18375	8851.386308	4329.509991	4329.700134	192.176183
148000	18500	8940.063099	4380.795276	4380.928855	178.338968
149000	18625	8832.407324	4336.139158	4336.310445	159.957721
150000	18750	8781.753063	4309.472448	4311.010204	161.270411
151000	18875	8805.501117	4330.44482	4330.643957	144.41234
152000	19000	8941.570952	4394.781555	4394.910711	151.878686
153000	19125	8969.101971	4401.732485	4402.002549	165.366937
154000	19250	8884.75344	4354.324866	4354.517432	175.911142
155000	19375	8919.171875	4386.650322	4387.066693	145.45486
156000	19500	8854.953478	4348.094547	4348.017425	158.841506
157000	19625	9012.721255	4431.793888	4432.148052	148.779315
158000	19750	9025.089807	4435.838015	4436.14167	153.110122
159000	19875	9046.161881	4435.919759	4436.030994	174.211128
160000	20000	8910.686904	4379.519194	4379.849761	151.317949
161000	20125	8980.886387	4393.723821	4393.834002	193.328564
162000	20250	9029.38966	4419.266556	4419.51124	190.611864
163000	20375	8988.412613	4421.526022	4421.82005	145.066541
164000	20500	9058.137613	4445.637825	4446.015368	166.48442
165000	20625	9062.866634	4460.103451	4460.272446	142.490737
166000	20750	8922.224326	4387.496199	4387.66403	147.064097
167000	20875	9180.962993	4516.563467	4516.69618	147.703346
168000	21000	9105.814442	4479.59921	4479.817397	146.397835
169000	21125	9036.632422	4432.080203	4432.355867	172.196352

Total Messages	Payload Size (kiB)	Input Rate (bytes/sec)	Output A Rate (bytes/sec)	Output B Rate (bytes/sec)	Delta Rate Input / Output (bytes/sec)
170000	21250	9116.952861	4481.707473	4482.038502	153.206886
171000	21375	8999.229855	4422.543064	4422.657549	154.029242
172000	21500	9050.448515	4438.982627	4439.158324	172.307564
173000	21625	9086.292414	4467.443163	4467.641316	151.207935
174000	21750	9182.073553	4512.333062	4512.466621	157.27387
175000	21875	9098.343264	4465.561371	4465.952394	166.829499
176000	22000	9165.776345	4515.189461	4515.34544	135.241444
177000	22125	9170.586747	4510.259382	4510.415452	149.911913
178000	22250	9208.239843	4527.503374	4527.649085	153.087384
179000	22375	9134.447328	4490.682792	4490.924111	152.840425
180000	22500	9154.688604	4513.225176	4513.44903	128.014398
181000	22625	9126.807102	4489.296682	4489.461084	148.049336
182000	22750	9121.928203	4488.300695	4488.499719	145.127789
183000	22875	9171.78896	4519.878892	4520.01161	131.898458
184000	23000	9181.791589	4520.728382	4521.127277	139.93593
185000	23125	9237.93295	4546.709356	4546.789331	144.434263
186000	23250	9261.271677	4559.77963	4560.058673	141.433374
187000	23375	9292.825889	4578.664255	4578.796375	135.365259
188000	23500	9374.928161	4611.45155	4611.634832	151.841779
189000	23625	9336.081138	4586.367091	4586.581212	163.132835
190000	23750	9363.874085	4607.0076	4607.177133	149.689352
191000	23875	9275.481009	4574.244924	4574.445309	126.790776
192000	24000	9321.783449	4595.644665	4595.966798	130.171986
193000	24125	9313.274374	4586.9249	4587.225789	139.123685
194000	24250	9186.794863	4509.29999	4509.411901	168.082972
195000	24375	9424.241663	4650.764124	4651.00873	122.468809
196000	24500	9346.971009	4599.256404	4599.441582	148.273023
197000	24625	9144.05262	4501.86286	4502.072915	140.116845
198000	24750	9378.251305	4622.898207	4623.176131	132.176967
199000	24875	9302.97006	4583.322072	4583.681875	135.966113
200000	25000	9246.151555	4554.644082	4555.040064	136.467409
201000	25125	9473.535819	4654.427678	4654.752837	164.355304
202000	25250	9391.021542	4627.139684	4627.367319	136.514539
203000	25375	9437.723231	4658.093656	4658.455783	121.173792

Total Messages	Payload Size (kiB)	Input Rate (bytes/sec)	Output A Rate (bytes/sec)	Output B Rate (bytes/sec)	Delta Rate Input / Output (bytes/sec)
204000	25500	9559.16863	4701.737194	4701.949637	155.481799
205000	25625	9413.106148	4634.282132	4634.444188	144.379828
206000	25750	9532.417786	4695.641901	4696.90365	139.872235
207000	25875	9445.88607	4653.132648	4653.452882	139.30054
208000	26000	9493.783261	4676.715432	4676.974247	140.093582
209000	26125	9552.638949	4707.687293	4707.856454	137.095202
210000	26250	9330.79204	4606.312018	4606.400219	118.079803
211000	26375	9436.231668	4658.591121	4658.806575	118.833972
212000	26500	9424.376844	4634.010586	4634.175283	156.190975
213000	26625	9403.101284	4622.018417	4622.253681	158.829186
214000	26750	9584.290792	4724.082346	4724.297671	135.910775
215000	26875	9518.69831	4682.049906	4682.173607	154.474797
216000	27000	9456.245732	4654.743217	4655.024911	146.477604
217000	27125	9571.710083	4724.073686	4724.109981	123.526416
218000	27250	9630.530017	4745.74167	4746.015583	138.772764
219000	27375	9522.306329	4691.745948	4691.965273	138.595108
220000	27500	9403.502899	4628.133602	4628.328868	147.040429
221000	27625	9400.907796	4642.621476	4642.802281	115.484039
222000	27750	9575.036173	4714.771704	4714.958132	145.306337
223000	27875	9723.476536	4797.44917	4797.664807	128.362559
224000	28000	9699.119878	4784.419036	4784.4493	130.251542
225000	28125	9712.21708	4790.410483	4790.589162	131.217435
226000	28250	9652.793296	4759.56704	4759.805594	133.420662
227000	28375	9606.668297	4743.797963	4744.00504	118.865294
228000	28500	9635.328592	4757.115184	4757.275232	120.938176
229000	28625	9600.515024	4725.011659	4724.903847	150.599518
230000	28750	9514.294048	4690.760802	4691.000328	132.532918
231000	28875	9732.283831	4792.500325	4792.660099	147.123407
232000	29000	9710.082062	4789.272625	4789.590872	131.218565
233000	29125	9622.297564	4738.019678	4738.249537	146.028349
234000	29250	9829.86974	4849.595875	4849.78677	130.487095
235000	29375	9716.538392	4802.821392	4803.041235	110.675765
236000	29500	9697.577352	4783.292467	4783.58276	130.702125
237000	29625	9898.816553	4880.650331	4880.830305	137.335917

Total Messages	Payload Size (kiB)	Input Rate (bytes/sec)	Output A Rate (bytes/sec)	Output B Rate (bytes/sec)	Delta Rate Input / Output (bytes/sec)
238000	29750	9593.271489	4720.28202	4720.348446	152.641023
239000	29875	9855.24013	4869.245898	4869.346809	116.647423
240000	30000	9788.127417	4831.040557	4831.179802	125.907058

B.5 Anycast Policy Throughput

Total Messages	Payload Size (kiB)	Input Rate (bytes/sec)	Output A Rate (bytes/sec)	Output B Rate (bytes/sec)	Std. Deviation (bytes/sec)	Delta Rate Input / Output (bytes/sec)
1000	125	940.584234	243.015246	255.57126	241.03669	246.5410653
2000	250	1779.332519	473.035157	508.69418	489.080036	490.269791
3000	375	2549.823552	663.963822	701.691535	702.407316	689.3542243
4000	500	3217.699924	879.949465	892.426758	904.190622	892.1889483
5000	625	3868.052977	1057.99909	1080.853049	1046.39486	1061.749
6000	750	4387.308686	1192.98359	1241.779278	1208.60888	1214.457249
7000	875	4873.966197	1350.65315	1399.07138	1367.360427	1372.361652
8000	1000	5422.651636	1520.657286	1559.198602	1527.170891	1535.675593
9000	1125	5714.695721	1580.368077	1604.982348	1591.818807	1592.389744
10000	1250	5992.13233	1695.322096	1716.066373	1689.352555	1700.247008
11000	1375	6511.268342	1872.190114	1879.979831	1881.877794	1878.015913
12000	1500	6855.470122	1968.069875	1954.420479	1944.177055	1955.555803
13000	1625	7173.284481	2006.793015	2012.779013	1999.720489	2006.430839
14000	1750	7405.381702	2138.89921	2168.599365	2151.19526	2152.897945
15000	1875	7624.783393	2206.709311	2239.77086	2231.681799	2226.05399
16000	2000	7799.372833	2272.670791	2291.08189	2287.503371	2283.752017
17000	2125	7863.286132	2318.814973	2331.723245	2327.286244	2325.941487
18000	2250	8412.012915	2495.714073	2503.904666	2491.80656	2497.141766

Total Messages	Payload Size (kiB)	Input Rate (bytes/sec)	Output A Rate (bytes/sec)	Output B Rate (bytes/sec)	Std. Deviation (bytes/sec)	Delta Rate Input / Output (bytes/sec)
19000	2375	8293.690946	2464.541184	2465.294287	2471.815631	2467.217034
20000	2500	8519.00142	2555.665874	2537.756489	2556.503107	2549.975157
21000	2625	8587.790942	2535.1481	2521.234404	2541.920665	2532.767723
22000	2750	9280.593688	2691.616622	2684.736751	2702.022975	2692.792116
23000	2875	9318.784685	2748.26664	2745.363964	2774.99896	2756.209855
24000	3000	9157.212163	2778.911137	2763.349002	2794.588853	2778.949664
25000	3125	9508.465577	2807.407585	2804.60045	2818.064398	2810.024144
26000	3250	9498.278985	2864.69511	2870.543431	2874.134087	2869.790876
27000	3375	9701.248263	2927.026584	2917.674937	2923.613575	2922.771699
28000	3500	9609.840477	2906.122133	2916.278251	2905.350108	2909.250164
29000	3625	9847.961065	3011.130389	3024.659722	3011.871902	3015.887338
30000	3750	9892.804864	3024.626366	3035.932495	3034.05362	3031.537494
31000	3875	10207.56928	3092.601902	3083.302498	3100.826767	3092.243722
32000	4000	10462.88113	3163.279704	3163.26424	3171.583377	3166.04244
33000	4125	9987.192183	3065.408143	3063.393566	3072.700244	3067.167318
34000	4250	9623.415922	2928.366627	2927.719343	2938.153547	2931.413172
35000	4375	10244.46214	3155.917488	3154.228566	3171.017164	3160.387739
36000	4500	10399.0761	3158.616864	3162.005814	3178.938497	3166.520392
37000	4625	10531.09748	3187.282693	3192.504428	3212.417602	3197.401574

Total Messages	Payload Size (kiB)	Input Rate (bytes/sec)	Output A Rate (bytes/sec)	Output B Rate (bytes/sec)	Std. Deviation (bytes/sec)	Delta Rate Input / Output (bytes/sec)
38000	4750	10266.66306	3164.827216	3167.341237	3187.41743	3173.195294
39000	4875	10347.71507	3178.259635	3174.953553	3185.712698	3179.641962
40000	5000	10615.28304	3304.844042	3304.183606	3311.11395	3306.713866
41000	5125	10456.34286	3199.484405	3195.775431	3197.056653	3197.43883
42000	5250	10661.45169	3298.58099	3304.937874	3297.111369	3300.210078
43000	5375	10423.81789	3256.780407	3262.663907	3257.197703	3258.880672
44000	5500	9663.539706	3019.32142	3024.66755	3015.927928	3019.972299
45000	5625	10253.57081	3194.421819	3201.103255	3188.278013	3194.601029
46000	5750	8000.185048	2379.794174	2385.040912	2380.211668	2381.682251
47000	5875	11181.63737	3463.358219	3473.368844	3461.170814	3465.965959
48000	6000	10855.8209	3402.643284	3404.354415	3392.632733	3399.876811
49000	6125	11369.74839	3515.905002	3519.792202	3503.443392	3513.046865
50000	6250	11472.58156	3598.58288	3602.909661	3575.018391	3592.170311
51000	6375	11220.95423	3499.693483	3509.91776	3484.498755	3498.036666
52000	6500	11269.5543	3564.257669	3573.706067	3543.507206	3560.490314
53000	6625	11335.46298	3582.490403	3586.818897	3553.673383	3574.327561
54000	6750	11192.87437	3533.211779	3531.803493	3509.804268	3524.939847
55000	6875	11419.65639	3624.672003	3623.036806	3598.565864	3615.424891
56000	7000	11392.20704	3594.338345	3591.905357	3572.30781	3586.183837

Total Messages	Payload Size (kiB)	Input Rate (bytes/sec)	Output A Rate (bytes/sec)	Output B Rate (bytes/sec)	Std. Deviation (bytes/sec)	Delta Rate Input / Output (bytes/sec)
57000	7125	11599.06076	3655.607012	3655.860296	3633.498286	3648.321865
58000	7250	11241.18367	3560.426088	3566.052141	3537.170021	3554.549417
59000	7375	11554.95862	3666.359597	3678.519758	3649.910537	3664.929964
60000	7500	11298.85413	3594.322394	3596.477199	3574.451871	3588.417155
61000	7625	11537.06851	3642.346485	3649.052252	3620.718208	3637.372315
62000	7750	11530.71121	3665.631448	3674.887853	3638.681514	3659.733605
63000	7875	11593.04226	3685.169221	3697.269452	3663.54644	3681.995038
64000	8000	11539.74866	3666.038856	3673.999146	3640.297134	3660.111712
65000	8125	11522.19751	3615.956725	3624.766256	3596.929933	3612.550971
66000	8250	11489.16259	3664.313835	3677.995478	3652.405355	3664.904889
67000	8375	11760.19385	3739.743473	3746.523975	3722.285837	3736.184428
68000	8500	11932.57743	3806.21123	3809.714175	3786.599681	3800.841695
69000	8625	12174.09378	3855.696878	3858.524565	3838.694679	3850.972041
70000	8750	12431.70333	3920.616394	3934.523782	3911.103024	3922.081067
71000	8875	11890.63032	3763.403272	3778.445042	3750.356614	3764.068309
72000	9000	11774.01736	3742.755937	3754.029372	3724.713385	3740.499565
73000	9125	11974.73364	3807.119715	3820.028904	3788.206876	3805.118498
74000	9250	11875.301	3790.422155	3809.252466	3772.393036	3790.689219
75000	9375	11794.2368	3778.179045	3786.935503	3755.330025	3773.481524

Total Messages	Payload Size (kiB)	Input Rate (bytes/sec)	Output A Rate (bytes/sec)	Output B Rate (bytes/sec)	Std. Deviation (bytes/sec)	Delta Rate Input / Output (bytes/sec)
76000	9500	11861.53716	3799.739057	3805.151299	3777.734346	3794.208234
77000	9625	11819.18429	3742.772585	3754.662256	3728.146151	3741.860331
78000	9750	11913.03849	3805.498254	3815.698047	3789.417216	3803.537839
79000	9875	11827.15821	3762.011537	3770.462441	3745.19081	3759.221596
80000	10000	12010.69552	3841.974032	3854.438649	3830.179439	3842.197373
81000	10125	12477.18677	3967.673819	3987.391171	3961.258716	3972.107902
82000	10250	12547.73879	3994.80476	4016.823464	3994.665957	4002.09806
83000	10375	11966.17868	3803.888827	3820.142966	3803.352332	3809.128042
84000	10500	11737.06952	3749.253597	3770.357342	3752.851171	3757.48737
85000	10625	12083.47549	3845.543689	3866.259376	3853.551674	3855.118246
86000	10750	11770.53157	3763.218523	3784.865052	3776.010708	3774.698094
87000	10875	11858.09217	3796.35885	3821.469519	3807.893968	3808.574112
88000	11000	12183.20551	3906.306466	3937.225536	3921.962925	3921.831642
89000	11125	12138.63356	3851.198947	3876.316823	3865.139238	3864.218336
90000	11250	12257.59082	3907.035883	3932.138429	3918.072712	3919.082341
91000	11375	12413.01681	3971.787124	3999.521416	3986.483711	3985.93075
92000	11500	12296.45195	3923.118206	3950.346444	3937.149235	3936.871295
93000	11625	12178.12272	3886.814333	3916.812538	3901.984017	3901.870296
94000	11750	12506.67078	3976.873106	4009.786355	3995.041466	3993.900309

Total Messages	Payload Size (kiB)	Input Rate (bytes/sec)	Output A Rate (bytes/sec)	Output B Rate (bytes/sec)	Std. Deviation (bytes/sec)	Delta Rate Input / Output (bytes/sec)
95000	11875	12395.23903	3949.344034	3980.778056	3965.959247	3965.360446
96000	12000	12218.45996	3904.080417	3931.01721	3914.728682	3916.60877
97000	12125	12472.96453	3998.465464	4021.45405	4010.267278	4010.062264
98000	12250	12489.37607	3988.392499	4011.413251	4001.866262	4000.557337
99000	12375	12522.67521	4011.933092	4035.322353	4023.269118	4023.508188
100000	12500	12177.35934	3915.283702	3943.017644	3922.419484	3926.906943
101000	12625	12290.15773	3911.539991	3939.723598	3921.499791	3924.25446
102000	12750	12212.52962	3920.891075	3944.895012	3928.008571	3931.264886
103000	12875	12152.33698	3898.22267	3921.944923	3912.831675	3910.999756
104000	13000	12193.33327	3906.988646	3930.309369	3921.067632	3919.455216
105000	13125	12479.84357	3980.864606	4003.696249	3995.149345	3993.236733
106000	13250	12355.83739	3981.694193	4001.265441	3998.107	3993.688878
107000	13375	12537.14422	4016.502564	4034.895648	4034.393759	4028.597324
108000	13500	12218.12337	3935.47192	3955.785271	3948.429218	3946.562136
109000	13625	12483.24068	3994.493128	4018.082865	4009.501778	4007.359257
110000	13750	12419.18357	3998.638165	4023.836283	4020.057092	4014.17718
111000	13875	12296.08031	3966.07929	3989.309543	3988.582618	3981.323817
112000	14000	12552.86634	4046.646637	4069.857791	4069.578836	4062.027755
113000	14125	12505.62615	4011.265518	4035.640895	4036.749418	4027.885277

Total Messages	Payload Size (kiB)	Input Rate (bytes/sec)	Output A Rate (bytes/sec)	Output B Rate (bytes/sec)	Std. Deviation (bytes/sec)	Delta Rate Input / Output (bytes/sec)
114000	14250	12509.47126	4021.268948	4047.703826	4050.46603	4039.812935
115000	14375	12385.60681	3986.600986	4014.267884	4017.355243	4006.074704
116000	14500	12456.96441	4012.853163	4039.251214	4044.745117	4032.283165
117000	14625	12590.99008	4036.735848	4060.368083	4069.026998	4055.376976
118000	14750	12472.09896	4022.655722	4045.150741	4057.853837	4041.886767
119000	14875	12346.23875	3986.472786	4004.707117	4018.662066	4003.280656
120000	15000	12746.64086	4100.596884	4121.644707	4133.624719	4118.622103
121000	15125	12586.86497	4046.027285	4070.959111	4081.719656	4066.235351
122000	15250	12800.03089	4115.534858	4140.48933	4153.743744	4136.589311
123000	15375	12931.32541	4163.961096	4189.664771	4204.754303	4186.126723
124000	15500	12633.01341	4070.286236	4097.911312	4106.722433	4091.639994
125000	15625	12658.25605	4061.302246	4088.037479	4098.254888	4082.531538
126000	15750	12536.51509	4046.845544	4077.715803	4085.812651	4070.124666
127000	15875	12389.43768	4002.943951	4028.171136	4036.035721	4022.383603
128000	16000	13040.04506	4196.35244	4221.452246	4234.057377	4217.287354
129000	16125	12977.31887	4174.605093	4198.061991	4214.311009	4195.659364
130000	16250	13052.33464	4204.306562	4234.460762	4251.733128	4230.166817
131000	16375	12944.35194	4167.188199	4199.975486	4217.112702	4194.758796
132000	16500	12926.434	4154.730768	4186.822363	4205.251081	4182.268071

Total Messages	Payload Size (kiB)	Input Rate (bytes/sec)	Output A Rate (bytes/sec)	Output B Rate (bytes/sec)	Std. Deviation (bytes/sec)	Delta Rate Input / Output (bytes/sec)
133000	16625	12685.32376	4093.108098	4119.584068	4137.170466	4116.620877
134000	16750	12753.85404	4110.246286	4140.614434	4157.562042	4136.140921
135000	16875	12781.46442	4117.112799	4152.943614	4167.875084	4145.977166
136000	17000	12857.12436	4145.104685	4185.741735	4200.622623	4177.156348
137000	17125	13292.14128	4265.085315	4302.108427	4313.648348	4293.61403
138000	17250	13002.00702	4188.594116	4225.337479	4240.768627	4218.233407
139000	17375	12492.80091	4010.337689	4047.572609	4059.954283	4039.288194
140000	17500	12568.84698	4043.525287	4076.867833	4092.935808	4071.109643
141000	17625	12620.20748	4055.545654	4087.42311	4101.228904	4081.399223
142000	17750	12710.19485	4104.83096	4136.641362	4149.729598	4130.40064
143000	17875	12748.19306	4114.5408	4144.274788	4156.49705	4138.437546
144000	18000	12541.74813	4045.619767	4072.437146	4084.900243	4067.652385
145000	18125	12706.59054	4098.01129	4123.817653	4132.89611	4118.241684
146000	18250	12612.80905	4084.361447	4108.796585	4121.429897	4104.862643
147000	18375	12684.80225	4105.747218	4129.984282	4142.4301	4126.053867
148000	18500	12612.07168	4066.646583	4093.736164	4105.383864	4088.58887
149000	18625	12665.33361	4075.683091	4104.978886	4115.005372	4098.555783
150000	18750	12984.15198	4173.096663	4202.139506	4212.906051	4196.047407
151000	18875	12379.14856	4003.07865	4033.310722	4038.78594	4025.058437

Total Messages	Payload Size (kiB)	Input Rate (bytes/sec)	Output A Rate (bytes/sec)	Output B Rate (bytes/sec)	Std. Deviation (bytes/sec)	Delta Rate Input / Output (bytes/sec)
152000	19000	12903.23786	4175.23231	4205.629657	4210.591318	4197.151095
153000	19125	13020.28262	4197.122332	4226.764994	4232.180337	4218.689221
154000	19250	12834.02316	4148.040087	4177.494308	4181.843499	4169.125965
155000	19375	10653.6931	2778.95394	2781.737675	2759.066633	2773.252749
156000	19500	11491.8744	3689.585594	3715.669747	3719.471418	3708.242253
157000	19625	12373.18178	3987.72694	4015.022333	4022.920306	4008.556526
158000	19750	12251.74039	3925.041713	3950.811528	3958.719987	3944.857743
159000	19875	12603.30449	4082.010439	4110.139235	4117.053657	4103.067777
160000	20000	12959.6767	4178.068871	4207.505742	4215.281124	4200.285246
161000	20125	12369.40197	3986.611185	4017.168375	4023.480667	4009.086742
162000	20250	12548.31877	4033.881128	4063.356398	4071.315815	4056.184447
163000	20375	12692.61036	4109.346446	4139.866114	4146.431721	4131.881427
164000	20500	12701.31094	4114.770636	4146.063911	4148.873573	4136.569373
165000	20625	12731.94714	4102.981186	4135.150882	4137.719476	4125.283848
166000	20750	13112.63453	4236.474448	4269.555973	4271.107824	4259.046082
167000	20875	12956.70846	4187.435112	4219.790643	4222.197291	4209.807682
168000	21000	12953.61257	4194.569603	4225.233305	4227.953992	4215.918967
169000	21125	12869.92368	4157.696837	4187.227081	4190.406956	4178.443625
170000	21250	12732.47057	4101.968089	4133.454752	4133.958636	4123.127159

Total Messages	Payload Size (kiB)	Input Rate (bytes/sec)	Output A Rate (bytes/sec)	Output B Rate (bytes/sec)	Std. Deviation (bytes/sec)	Delta Rate Input / Output (bytes/sec)
171000	21375	12756.32854	4134.228761	4167.719426	4166.624918	4156.191035
172000	21500	12690.40683	4115.326329	4147.325066	4146.474134	4136.375176
173000	21625	12719.48837	4111.785184	4143.753536	4141.510809	4132.349843
174000	21750	12542.92528	4044.222682	4074.255783	4072.747695	4063.742053
175000	21875	12582.59304	4087.486974	4114.280438	4113.63944	4105.135617
176000	22000	12761.25312	4143.323693	4169.657849	4168.700523	4160.560688
177000	22125	12495.07609	4055.799163	4078.672576	4079.824439	4071.432059
178000	22250	12978.1614	4208.036542	4230.100745	4231.49389	4223.210392
179000	22375	12812.45113	4167.476584	4192.162514	4190.901058	4183.513385
180000	22500	12760.02811	4149.999958	4172.507888	4171.822012	4164.776619
181000	22625	12832.3981	4162.097344	4182.717804	4183.431148	4176.082099
182000	22750	12983.97314	4221.63847	4245.41702	4243.885353	4236.980281
183000	22875	12447.0431	3932.266373	3953.816018	3954.284454	3946.788948
184000	23000	13385.12796	4350.702909	4377.530017	4375.959014	4368.06398
185000	23125	12786.86732	4143.232573	4172.947163	4172.781395	4162.987044
186000	23250	12673.56999	4110.662319	4136.580218	4137.406947	4128.216495
187000	23375	12556.30614	4067.576888	4094.260043	4095.053569	4085.630167
188000	23500	12978.34343	4219.89746	4246.359334	4246.488584	4237.581793
189000	23625	12780.19219	4147.779688	4174.848629	4173.485513	4165.371277

Total Messages	Payload Size (kiB)	Input Rate (bytes/sec)	Output A Rate (bytes/sec)	Output B Rate (bytes/sec)	Std. Deviation (bytes/sec)	Delta Rate Input / Output (bytes/sec)
190000	23750	12828.05038	4170.115051	4198.715066	4194.835337	4187.888485
191000	23875	12679.63929	4108.323389	4136.93426	4133.511508	4126.256386
192000	24000	12868.76122	4188.909995	4214.816752	4214.850961	4206.192569
193000	24125	13019.21103	4220.951703	4245.040898	4243.590096	4236.527566
194000	24250	13027.09979	4233.886806	4260.540245	4256.539804	4250.322285
195000	24375	12691.52074	4108.02103	4134.685084	4130.675271	4124.460462
196000	24500	12930.38216	4210.62491	4236.113629	4231.608714	4226.115751
197000	24625	12575.2432	4075.370665	4100.010363	4095.771168	4090.384065
198000	24750	12670.01415	4126.11182	4148.983928	4146.597585	4140.564444
199000	24875	12806.87015	4175.927396	4197.293689	4196.96925	4190.063445
200000	25000	12522.92086	4064.339192	4085.204245	4084.6663	4078.069912
201000	25125	12736.89758	4135.453994	4154.201577	4159.813737	4149.823103
202000	25250	12711.53057	4151.481261	4170.48468	4175.11593	4165.693957
203000	25375	13026.2041	4226.625985	4246.092412	4250.355116	4241.024504
204000	25500	13157.62571	4280.49161	4298.850036	4304.781387	4294.707678
205000	25625	12609.05293	4107.719574	4123.331804	4130.669842	4120.57374
206000	25750	12814.58404	4183.741251	4198.452132	4207.464555	4196.552646
207000	25875	12841.28985	4188.158385	4202.035992	4209.587126	4199.927168
208000	26000	13096.61949	4254.941867	4269.315941	4275.748321	4266.66871

Total Messages	Payload Size (kiB)	Input Rate (bytes/sec)	Output A Rate (bytes/sec)	Output B Rate (bytes/sec)	Std. Deviation (bytes/sec)	Delta Rate Input / Output (bytes/sec)
209000	26125	12967.24022	4227.530952	4242.962846	4247.769354	4239.421051
210000	26250	13019.09132	4230.328049	4244.804516	4250.3772	4241.836588
211000	26375	12608.05055	4102.012819	4117.585378	4122.302567	4113.966921
212000	26500	12277.53341	4005.755467	4021.572027	4023.555266	4016.96092
213000	26625	13077.8603	4263.130437	4279.986119	4279.350924	4274.155827
214000	26750	12755.72699	4170.581917	4191.947626	4189.562687	4184.030743
215000	26875	13078.46189	4266.964647	4286.868118	4285.166626	4279.666464
216000	27000	12601.86361	4108.45165	4127.569404	4126.973071	4120.998042
217000	27125	12596.37679	4107.064855	4129.917585	4127.264933	4121.415791
218000	27250	13319.62512	4346.663289	4370.706595	4367.150742	4361.506875
219000	27375	12748.50266	4165.494555	4187.597777	4185.422884	4179.505072
220000	27500	12926.01198	4218.424106	4239.29141	4237.800235	4231.838584
221000	27625	12851.20123	4184.169199	4204.952624	4205.219747	4198.113857
222000	27750	12656.53122	4131.336229	4153.292983	4154.062996	4146.230736
223000	27875	12430.79309	4059.310723	4084.144049	4083.441587	4075.63212
224000	28000	13251.82358	4319.000174	4346.042058	4348.04794	4337.696724
225000	28125	12911.38617	4199.894174	4226.234582	4230.666543	4218.931766
226000	28250	12789.8022	4175.011011	4202.541197	4206.430733	4194.66098
227000	28375	13015.28935	4241.587826	4272.544303	4276.55206	4263.561396

Total Messages	Payload Size (kiB)	Input Rate (bytes/sec)	Output A Rate (bytes/sec)	Output B Rate (bytes/sec)	Std. Deviation (bytes/sec)	Delta Rate Input / Output (bytes/sec)
228000	28500	12991.77797	4231.371492	4263.466645	4266.15805	4253.665396
229000	28625	13078.57563	4253.946596	4282.940731	4288.508101	4275.131809
230000	28750	13044.22417	4253.774357	4282.547132	4289.650343	4275.323944
231000	28875	13289.10153	4323.014739	4354.392229	4359.761751	4345.722906
232000	29000	13113.30267	4280.861165	4310.052385	4315.165571	4302.026374
233000	29125	13014.35176	4244.479232	4275.765477	4278.993477	4266.412729
234000	29250	13000.00939	4220.282757	4251.790694	4255.083698	4242.385716
235000	29375	13121.36064	4279.002839	4308.845538	4313.239854	4300.362744
236000	29500	13018.31009	4251.943253	4280.806079	4283.551965	4272.100432
237000	29625	12557.00257	4086.963689	4113.960732	4116.390038	4105.771486
238000	29750	13003.83225	4247.623757	4273.577686	4277.863591	4266.355011
239000	29875	13255.52176	4332.41578	4358.184299	4361.545827	4350.715302
240000	30000	13076.3297	4272.80629	4297.602167	4300.804995	4290.404484
241000	30125	13032.54649	4248.372653	4273.014868	4277.304629	4266.230717
242000	30250	12802.22374	4179.086124	4203.790352	4205.812549	4196.229675
243000	30375	12919.70705	4213.102587	4237.533017	4240.527968	4230.387857
244000	30500	13089.91878	4276.71668	4302.707412	4304.332726	4294.585606
245000	30625	13242.26976	4311.864178	4337.743431	4336.611028	4328.739546
246000	30750	13067.87208	4271.607681	4297.57412	4295.666046	4288.282616

Total Messages	Payload Size (kiB)	Input Rate (bytes/sec)	Output A Rate (bytes/sec)	Output B Rate (bytes/sec)	Std. Deviation (bytes/sec)	Delta Rate Input / Output (bytes/sec)
247000	30875	13244.57644	4325.916814	4352.016046	4351.477777	4343.136879
248000	31000	13056.21173	4258.214383	4286.040517	4286.131452	4276.795451
249000	31125	12876.30018	4202.930566	4232.377779	4229.901666	4221.73667
250000	31250	12934.13288	4227.296709	4255.706715	4253.424211	4245.475878
251000	31375	13182.17977	4311.907644	4340.067214	4336.815499	4329.596786
252000	31500	13564.18865	4431.67549	4459.125961	4456.457343	4449.086265
253000	31625	13047.69572	4256.841715	4283.503287	4281.327964	4273.890989
254000	31750	13094.5965	4284.330769	4311.340807	4308.722391	4301.464656
255000	31875	13126.2745	4292.920942	4321.843705	4317.482076	4310.748908
256000	32000	13557.02064	4427.972591	4457.800216	4453.990974	4446.587927
257000	32125	13371.17915	4355.339264	4385.177108	4382.189398	4374.235257
258000	32250	13335.27746	4361.304796	4392.710178	4388.018476	4380.677817
259000	32375	13513.91401	4420.909574	4451.267321	4447.718406	4439.9651
260000	32500	13214.90945	4317.584629	4344.981308	4340.694967	4334.420301
261000	32625	13550.05109	4419.252207	4447.54521	4443.678849	4436.825422
262000	32750	13529.63351	4420.619432	4448.00521	4446.522224	4438.382289
263000	32875	13450.98186	4394.41831	4423.571283	4421.252042	4413.080545
264000	33000	13488.11146	4411.798059	4439.160458	4435.346556	4428.768358
265000	33125	13216.76776	4318.503944	4344.495857	4340.642418	4334.547406

Total Messages	Payload Size (kiB)	Input Rate (bytes/sec)	Output A Rate (bytes/sec)	Output B Rate (bytes/sec)	Std. Deviation (bytes/sec)	Delta Rate Input / Output (bytes/sec)
266000	33250	12811.20944	4189.945329	4213.649174	4210.095517	4204.56334
267000	33375	12919.0577	4222.852861	4248.378326	4243.860558	4238.363915
268000	33500	13048.75111	4268.337041	4295.390557	4288.31097	4284.012856
269000	33625	13118.39837	4279.18909	4307.751463	4298.995208	4295.31192
270000	33750	12754.14223	4174.138775	4201.650306	4191.491591	4189.093557
271000	33875	13157.12177	4311.423173	4338.761319	4327.837064	4326.007185
272000	34000	13215.5491	4328.099971	4355.290151	4343.461169	4342.283764
273000	34125	13354.76357	4359.715203	4387.130301	4375.485555	4374.110353
274000	34250	13111.54093	4295.063512	4322.498471	4309.381714	4308.981232
275000	34375	13209.131	4323.020619	4352.289601	4338.876898	4338.062373
276000	34500	13314.61312	4362.375807	4391.568524	4377.262105	4377.068812
277000	34625	13184.1842	4316.6013	4346.034714	4330.079785	4330.905266
278000	34750	12951.95389	4244.350257	4271.890499	4257.043434	4257.761397
279000	34875	13087.43037	4289.181777	4319.462522	4303.734548	4304.126282
280000	35000	12746.60643	4176.270895	4207.951461	4192.882678	4192.368345
281000	35125	13362.6589	4364.789055	4396.581801	4380.423479	4380.598112
282000	35250	13003.33992	4263.490524	4296.806768	4278.380895	4279.559396
283000	35375	13168.16331	4311.960548	4346.908371	4328.177286	4329.015402
284000	35500	13016.31994	4258.413878	4292.453489	4276.613825	4275.827064

Total Messages	Payload Size (kiB)	Input Rate (bytes/sec)	Output A Rate (bytes/sec)	Output B Rate (bytes/sec)	Std. Deviation (bytes/sec)	Delta Rate Input / Output (bytes/sec)
285000	35625	13102.60067	4283.294232	4319.152396	4301.317803	4301.25481
286000	35750	13266.98296	4347.575407	4384.813923	4368.235099	4366.87481
287000	35875	13239.44002	4327.953394	4367.048789	4349.20666	4348.069614
288000	36000	13341.30724	4363.915718	4402.78176	4386.272074	4384.323184
289000	36125	13299.93715	4344.374114	4383.384174	4365.110088	4364.289459
290000	36250	12886.50746	4214.626833	4252.268538	4236.194569	4234.363313
291000	36375	13126.92869	4297.535799	4335.921208	4318.912448	4317.456485
292000	36500	13858.35161	4525.778385	4564.582476	4548.356309	4546.239057
293000	36625	13522.28785	4415.490659	4454.597051	4436.079855	4435.389188
294000	36750	13275.94692	4343.474368	4381.499532	4363.115711	4362.696537
295000	36875	13831.83842	4517.925904	4556.971433	4538.156876	4537.684738
296000	37000	13615.32449	4454.830876	4493.403446	4473.7455	4473.993274
297000	37125	13241.37134	4332.21802	4368.897936	4349.708417	4350.274791
298000	37250	13335.21816	4369.0928	4408.17516	4387.751595	4388.339852
299000	37375	13115.41911	4291.383724	4330.712246	4310.777064	4310.957678
300000	37500	13424.74811	4394.293814	4433.172703	4414.234935	4413.900484
301000	37625	13377.32176	4370.342466	4409.533754	4389.558077	4389.811432
302000	37750	12896.11187	4225.452699	4263.925131	4243.921477	4244.433102
303000	37875	13285.34357	4348.105495	4387.415131	4366.817147	4367.445924

Total Messages	Payload Size (kiB)	Input Rate (bytes/sec)	Output A Rate (bytes/sec)	Output B Rate (bytes/sec)	Std. Deviation (bytes/sec)	Delta Rate Input / Output (bytes/sec)
304000	38000	13461.98325	4409.574125	4448.008982	4428.236269	4428.606459
305000	38125	13215.52709	4324.335559	4361.468542	4340.794102	4342.199401
306000	38250	13047.85536	4278.616633	4316.534991	4292.383795	4295.84514
307000	38375	13218.81676	4336.391447	4372.291999	4349.672185	4352.78521
308000	38500	13023.31584	4268.389306	4302.797815	4280.502577	4283.896566
309000	38625	13130.35755	4296.122182	4329.05441	4307.219996	4310.798863
310000	38750	13225.49069	4324.772015	4358.555969	4334.580849	4339.302944
311000	38875	12888.4436	4229.920341	4262.134448	4236.659465	4242.904751
312000	39000	13063.02912	4290.841044	4324.181622	4298.079508	4304.367391
313000	39125	12978.32164	4250.510224	4283.339302	4257.974992	4263.941506
314000	39250	13047.36935	4268.305693	4300.388296	4275.498281	4281.397423
315000	39375	12982.66509	4250.784674	4281.993197	4258.845344	4263.874405
316000	39500	13093.82246	4295.370525	4324.390896	4302.314368	4307.358596
317000	39625	13151.49968	4306.944188	4336.62889	4312.258462	4318.610513
318000	39750	13042.24465	4267.222464	4296.12511	4271.90223	4278.416601
319000	39875	12972.51945	4245.080442	4274.922815	4251.797388	4257.266882
320000	40000	13400.81481	4385.440552	4415.486462	4391.672862	4397.533292
321000	40125	12300.67053	3750.125411	3771.986857	3770.534024	3764.215431
322000	40250	12881.84983	4227.123997	4254.939934	4231.690017	4237.917983

Total Messages	Payload Size (kiB)	Input Rate (bytes/sec)	Output A Rate (bytes/sec)	Output B Rate (bytes/sec)	Std. Deviation (bytes/sec)	Delta Rate Input / Output (bytes/sec)
323000	40375	12829.79066	4211.91461	4239.87555	4217.042517	4222.944226
324000	40500	13232.56598	4344.371531	4372.813457	4349.204675	4355.463221
325000	40625	13070.60847	4282.002893	4309.799002	4286.502552	4292.768149
326000	40750	12771.5727	4173.86777	4200.704724	4177.498904	4184.023799
327000	40875	13150.41468	4309.58855	4338.133335	4314.007176	4320.576354
328000	41000	13050.12286	4279.570317	4307.23489	4283.67298	4290.159396
329000	41125	12759.37997	4186.095478	4212.52145	4189.149521	4195.92215
330000	41250	12699.73069	4155.270524	4180.620872	4158.051781	4164.647726
331000	41375	13161.97685	4317.747877	4342.545269	4319.549484	4326.61421
332000	41500	13258.35441	4350.689341	4377.07265	4353.400546	4360.387512
333000	41625	13176.51252	4319.809599	4344.864157	4320.133477	4328.269078
334000	41750	12463.32927	4036.387667	4060.908671	4038.716395	4045.337578
335000	41875	13037.4428	4282.720256	4307.618593	4283.694502	4291.34445
336000	42000	12976.03311	4259.001049	4283.499731	4259.803003	4267.434594
337000	42125	12965.99926	4245.262493	4270.138267	4246.603679	4254.00148
338000	42250	13227.72702	4329.713279	4354.412682	4330.545689	4338.223883
339000	42375	13316.71376	4375.265074	4400.784146	4377.922948	4384.657389
340000	42500	13277.48355	4359.828487	4386.826097	4364.965246	4370.539943
341000	42625	13042.00827	4280.977117	4307.716938	4284.572648	4291.088901

Total Messages	Payload Size (kiB)	Input Rate (bytes/sec)	Output A Rate (bytes/sec)	Output B Rate (bytes/sec)	Std. Deviation (bytes/sec)	Delta Rate Input / Output (bytes/sec)
342000	42750	12964.51487	4255.968433	4283.950061	4261.270701	4267.063065
343000	42875	13237.91741	4351.117523	4377.39271	4355.491581	4361.333938
344000	43000	12292.33019	4038.634201	4062.585822	4042.497634	4047.905886
345000	43125	13193.23066	4333.917131	4357.927389	4338.525587	4343.456702
346000	43250	13471.57191	4414.435165	4438.973584	4419.044024	4424.150924
347000	43375	13448.02835	4411.483589	4434.539534	4415.612536	4420.54522
348000	43500	13147.0517	4324.945612	4347.607921	4330.577399	4334.376977
349000	43625	13201.25072	4333.749433	4355.924615	4339.346553	4343.006867
350000	43750	13469.73376	4431.770256	4452.365689	4434.915787	4439.683911
351000	43875	13538.66382	4451.86188	4471.242658	4454.164495	4459.089678
352000	44000	13139.85102	4327.221762	4345.837577	4329.915592	4334.324977
353000	44125	13615.3251	4474.887952	4494.223693	4477.86104	4482.324228
354000	44250	13276.83005	4368.645833	4388.796631	4372.14013	4376.527531
355000	44375	13392.19663	4401.539344	4421.792993	4404.925076	4409.419138
356000	44500	13694.33814	4499.307307	4520.665233	4501.730133	4507.234224
357000	44625	13241.22632	4350.941149	4371.196903	4353.078032	4358.405361
358000	44750	13581.69576	4467.124059	4487.566857	4468.503853	4474.398256
359000	44875	13249.88	4361.756628	4381.658125	4362.840859	4368.751871
360000	45000	13238.82285	4353.582406	4373.378785	4355.025668	4360.662286

Total Messages	Payload Size (kiB)	Input Rate (bytes/sec)	Output A Rate (bytes/sec)	Output B Rate (bytes/sec)	Std. Deviation (bytes/sec)	Delta Rate Input / Output (bytes/sec)
361000	45125	13358.78484	4392.976414	4412.402709	4394.868978	4400.0827
362000	45250	13341.91786	4393.27066	4411.919574	4396.671661	4400.620632
363000	45375	13422.16613	4412.653852	4431.928499	4415.740237	4420.107529
364000	45500	13504.87443	4445.120141	4464.142477	4449.390662	4452.884427
365000	45625	13345.94068	4386.92668	4405.269295	4389.689113	4393.961696
366000	45750	13328.42197	4388.290383	4406.119561	4390.720663	4395.043536
367000	45875	13445.08901	4421.42193	4438.986246	4423.249734	4427.88597
368000	46000	13358.48455	4396.417699	4414.730108	4399.573212	4403.573673
369000	46125	13603.37671	4473.665421	4493.468151	4476.51508	4481.216217
370000	46250	13410.69492	4415.236509	4434.701129	4417.72864	4422.555426
371000	46375	12926.53335	4259.186452	4278.362913	4260.239358	4265.929574
372000	46500	13430.38641	4413.61793	4433.475994	4412.742649	4419.945524
373000	46625	13290.67916	4370.085684	4388.612895	4368.863621	4375.854067
374000	46750	13322.20015	4388.429992	4407.39711	4386.706423	4394.177842
375000	46875	13343.79914	4394.368847	4413.046481	4391.193171	4399.536166
376000	47000	13501.19484	4446.725285	4465.551238	4443.441129	4451.905884
377000	47125	13658.68292	4494.809652	4513.811272	4490.889474	4499.836799
378000	47250	13768.40723	4532.149162	4551.002337	4528.307359	4537.152953
379000	47375	13313.36773	4387.017467	4406.204771	4383.666003	4392.29608

Total Messages	Payload Size (kiB)	Input Rate (bytes/sec)	Output A Rate (bytes/sec)	Output B Rate (bytes/sec)	Std. Deviation (bytes/sec)	Delta Rate Input / Output (bytes/sec)
380000	47500	13294.05384	4381.578727	4399.761642	4376.522497	4385.954289
381000	47625	13297.41148	4372.925483	4392.071856	4368.204706	4377.734015
382000	47750	13310.37027	4375.009258	4392.916671	4370.151742	4379.359224
383000	47875	13446.19288	4422.783533	4440.907585	4417.478236	4427.056451
384000	48000	14094.43891	4638.335332	4657.75519	4632.825585	4642.972036
385000	48125	13544.3103	4458.448719	4477.160545	4452.102993	4462.570752
386000	48250	13277.79363	4373.365877	4392.180247	4367.168168	4377.571431
387000	48375	13363.22091	4403.632902	4421.860126	4396.581558	4407.358195
388000	48500	13377.79819	4408.915461	4428.475596	4403.813257	4413.734771
389000	48625	13113.53255	4311.160161	4330.842363	4307.113995	4316.372173
390000	48750	13835.08676	4559.212525	4579.425	4554.225923	4564.287816
391000	48875	13525.64424	4451.42686	4470.248743	4445.947999	4455.874534
392000	49000	13861.64445	4560.254113	4580.562624	4555.063283	4565.29334
393000	49125	13498.45505	4437.071315	4457.297315	4432.810211	4442.392947
394000	49250	13102.94368	4317.010082	4335.245496	4312.651871	4321.635816
395000	49375	13485.95474	4442.642534	4461.34167	4438.452404	4447.478869
396000	49500	12990.91214	4285.712055	4302.580833	4281.432324	4289.908404
397000	49625	13268.9636	4367.57329	4384.089881	4364.132845	4371.932005
398000	49750	13399.33482	4411.722826	4428.950478	4408.606726	4416.426677

Total Messages	Payload Size (kiB)	Input Rate (bytes/sec)	Output A Rate (bytes/sec)	Output B Rate (bytes/sec)	Std. Deviation (bytes/sec)	Delta Rate Input / Output (bytes/sec)
399000	49875	13137.8085	4330.173254	4346.902706	4327.745287	4334.940416
400000	50000	13344.09846	4396.872588	4413.677927	4394.388969	4401.646495
401000	50125	13463.48619	4431.490156	4450.618545	4431.399388	4437.83603
402000	50250	13616.55589	4487.542752	4505.405274	4486.563933	4493.170653
403000	50375	13683.34399	4508.531406	4527.122208	4507.824422	4514.492679
404000	50500	13707.4206	4513.122293	4531.935663	4512.180082	4519.079346
405000	50625	13162.02594	4331.142545	4349.843454	4330.442822	4337.14294
406000	50750	13051.18682	4300.550724	4320.189311	4300.695487	4307.145174
407000	50875	13354.62987	4398.193455	4418.452375	4398.059954	4404.901928
408000	51000	13097.63035	4318.7073	4337.556624	4317.142208	4324.468711
409000	51125	13398.81916	4411.788631	4431.903051	4411.089554	4418.260412
410000	51250	13100.54077	4308.125648	4329.254077	4309.542752	4315.640826
411000	51375	13121.39922	4320.976115	4341.987918	4322.926443	4328.630159
412000	51500	12977.84741	4278.660789	4298.608885	4279.23533	4285.501668
413000	51625	13476.55155	4429.261405	4449.800518	4430.040444	4436.367456
414000	51750	13376.43691	4404.6317	4426.420474	4404.942432	4411.998202
415000	51875	13665.41229	4502.438599	4524.07909	4502.228325	4509.582005
416000	52000	13227.73951	4356.953611	4377.213007	4356.966287	4363.710968
417000	52125	13278.52603	4370.935327	4390.719153	4372.156332	4377.936937

Total Messages	Payload Size (kiB)	Input Rate (bytes/sec)	Output A Rate (bytes/sec)	Output B Rate (bytes/sec)	Std. Deviation (bytes/sec)	Delta Rate Input / Output (bytes/sec)
418000	52250	13464.09872	4435.732837	4455.322616	4437.486095	4442.847183
419000	52375	13359.34505	4403.150748	4422.281297	4405.048808	4410.160284
420000	52500	13713.27552	4516.018615	4536.686902	4519.195717	4523.967078
421000	52625	13368.6299	4401.308763	4421.264652	4403.672773	4408.748729
422000	52750	13263.10134	4370.149033	4389.075889	4372.744423	4377.323115
423000	52875	13340.56268	4390.013766	4409.923461	4391.255277	4397.064168
424000	53000	13388.4618	4412.849142	4433.165189	4413.768062	4419.927464
425000	53125	13639.0347	4492.061261	4513.471003	4492.426006	4499.319423
426000	53250	13508.31527	4449.854542	4469.647647	4449.424019	4456.308736
427000	53375	13503.8128	4450.592536	4473.036133	4451.509306	4458.379325
428000	53500	13320.95315	4377.357681	4398.28952	4377.550074	4384.399092
429000	53625	12863.90727	4231.821328	4253.125744	4232.41116	4239.119411
430000	53750	12855.17119	4217.408202	4239.790664	4219.297154	4225.498673
431000	53875	13341.50746	4392.786443	4416.995342	4395.407292	4401.729692
432000	54000	12769.43508	4199.462583	4222.822729	4201.748801	4208.011371
433000	54125	13006.45264	4278.039325	4301.340505	4280.598195	4286.659342
434000	54250	13030.84831	4279.489705	4303.782233	4281.932933	4288.401624
435000	54375	11943.25314	3527.973495	3557.213853	3570.654608	3551.947319
436000	54500	13039.01135	4282.208024	4307.129542	4285.779631	4291.705732

Total Messages	Payload Size (kiB)	Input Rate (bytes/sec)	Output A Rate (bytes/sec)	Output B Rate (bytes/sec)	Std. Deviation (bytes/sec)	Delta Rate Input / Output (bytes/sec)
437000	54625	13412.91088	4415.998365	4440.433983	4417.512388	4424.648245
438000	54750	12990.72	4276.449567	4301.459698	4279.046982	4285.652082
439000	54875	13304.84016	4379.572975	4405.801837	4382.430305	4389.268372
440000	55000	12894.48629	4240.664493	4266.464441	4244.284183	4250.471039
441000	55125	12876.17031	4231.199559	4258.019105	4234.467935	4241.228866
442000	55250	13047.15087	4285.963023	4314.30873	4290.795273	4297.022342
443000	55375	13447.46171	4418.913493	4448.614726	4422.924996	4430.151072
444000	55500	12859.43273	4233.943515	4261.916226	4236.117175	4243.992305
445000	55625	12447.23074	4096.08127	4122.618461	4097.538996	4105.412909
446000	55750	11396.45722	3167.814042	3194.385733	3181.029526	3181.076434
447000	55875	12517.32263	4115.215125	4141.944991	4116.383946	4124.514687
448000	56000	12838.17266	4224.283369	4252.048118	4224.841591	4233.724359
449000	56125	12623.72729	4148.882072	4176.135736	4149.6318	4158.216536
450000	56250	12901.23375	4240.353159	4269.655083	4242.608142	4250.872128
451000	56375	13389.94956	4385.593408	4415.862648	4387.239171	4396.231742
452000	56500	13332.86688	4386.279159	4416.936965	4389.627673	4397.614599
453000	56625	12401.76426	3831.090521	3857.533583	3859.640145	3849.421416
454000	56750	13400.09216	4405.275398	4437.488134	4408.505253	4417.089595
455000	56875	12969.16378	4273.807537	4304.3959	4276.61884	4284.940759

Total Messages	Payload Size (kiB)	Input Rate (bytes/sec)	Output A Rate (bytes/sec)	Output B Rate (bytes/sec)	Std. Deviation (bytes/sec)	Delta Rate Input / Output (bytes/sec)
456000	57000	13272.0846	4371.694848	4403.144453	4374.983484	4383.274262
457000	57125	13613.40811	4479.600336	4511.6436	4482.427698	4491.223878
458000	57250	13306.44502	4374.050017	4405.284441	4377.221533	4385.518664
459000	57375	13359.29752	4394.754451	4426.228046	4398.253432	4406.411976
460000	57500	13554.30464	4465.47092	4497.700301	4469.078013	4477.416411
461000	57625	13354.62928	4396.400669	4427.482477	4400.010261	4407.964469
462000	57750	13734.57681	4522.432108	4555.186157	4526.680743	4534.766336
463000	57875	13259.23815	4367.605368	4399.257751	4371.786566	4379.549895
464000	58000	13448.10713	4427.572918	4460.446611	4432.069687	4440.029739
465000	58125	13343.4694	4386.392381	4418.597718	4390.659971	4398.550023
466000	58250	13410.4401	4412.1129	4442.334295	4414.619447	4423.022214
467000	58375	13189.52403	4346.288592	4376.47816	4349.480668	4357.415807
468000	58500	13258.37479	4368.678495	4399.2366	4372.266876	4380.060657
469000	58625	13099.97721	4313.303375	4342.845272	4317.741093	4324.629913
470000	58750	13464.29412	4428.22288	4458.119535	4432.904059	4439.748825
471000	58875	13545.64157	4464.135737	4493.661434	4468.196448	4475.331206
472000	59000	13282.3019	4377.208068	4405.879809	4381.677219	4388.255032
473000	59125	13439.37693	4426.887447	4455.139883	4431.265726	4437.764352
474000	59250	13247.5855	4364.943598	4394.197032	4369.95995	4376.36686

Total Messages	Payload Size (kiB)	Input Rate (bytes/sec)	Output A Rate (bytes/sec)	Output B Rate (bytes/sec)	Std. Deviation (bytes/sec)	Delta Rate Input / Output (bytes/sec)
475000	59375	13333.86931	4392.345073	4422.451446	4398.555219	4404.450579
476000	59500	13493.31415	4442.313734	4472.429119	4447.646099	4454.129651
477000	59625	13630.62693	4488.909365	4520.155984	4494.430707	4501.165352
478000	59750	13230.63262	4352.787935	4383.51773	4358.171638	4364.825768
479000	59875	13592.26706	4478.228814	4509.611348	4484.262997	4490.701053
480000	60000	13255.87813	4367.987164	4399.287915	4374.783632	4380.686237
481000	60125	13344.6105	4391.280704	4420.981704	4397.160967	4403.141125
482000	60250	13564.0467	4468.050822	4498.001423	4473.375374	4479.809206
483000	60375	12927.39943	4252.509427	4279.905859	4257.324648	4263.246645
484000	60500	13537.59059	4457.111494	4484.544621	4460.93899	4467.531702
485000	60625	13640.83085	4496.542751	4525.283715	4499.81122	4507.212562
486000	60750	13830.15736	4557.586742	4586.58183	4561.191652	4568.453408
487000	60875	13548.85076	4465.290199	4494.738879	4469.294483	4476.441187
488000	61000	13726.93327	4521.732218	4551.102729	4524.899382	4532.57811
489000	61125	13327.87427	4392.376583	4420.880371	4396.402255	4403.219736
490000	61250	13522.95273	4447.933005	4477.68692	4452.668076	4459.429334
491000	61375	13267.8168	4371.403044	4400.882644	4376.425478	4382.903722
492000	61500	13776.00568	4537.187884	4567.627667	4542.081679	4548.965743
493000	61625	13988.47083	4609.009984	4640.143545	4612.856477	4620.670002

Total Messages	Payload Size (kiB)	Input Rate (bytes/sec)	Output A Rate (bytes/sec)	Output B Rate (bytes/sec)	Std. Deviation (bytes/sec)	Delta Rate Input / Output (bytes/sec)
494000	61750	13523.53519	4458.782295	4488.741076	4462.044442	4469.855938
495000	61875	13488.63301	4445.61384	4474.803463	4448.211174	4456.209492
496000	62000	13447.90187	4426.295185	4455.433192	4429.163579	4436.963985
497000	62125	13866.25547	4567.188583	4597.285038	4569.925939	4578.133187
498000	62250	13346.96761	4400.893921	4430.002251	4403.992704	4411.629625
499000	62375	13531.09705	4458.027841	4487.584235	4461.659225	4469.090434
500000	62500	13694.33724	4512.325696	4542.380427	4516.484763	4523.730295
501000	62625	13631.47909	4495.012346	4524.465357	4497.965267	4505.814323
502000	62750	13612.23926	4487.887164	4517.247029	4491.288082	4498.807425
503000	62875	13966.02325	4601.518861	4632.738206	4605.172638	4613.143235
504000	63000	13502.98078	4446.051141	4476.569492	4449.983341	4457.534658
505000	63125	13525.35408	4459.236863	4490.032152	4463.2532	4470.840738
506000	63250	13342.38541	4390.004015	4419.720008	4392.854407	4400.859477
507000	63375	13469.85264	4439.423357	4469.55283	4442.932801	4450.636329
508000	63500	13396.1631	4418.055597	4448.429597	4421.363829	4429.283008
509000	63625	13733.63525	4519.543476	4551.164945	4523.462689	4531.39037
510000	63750	13666.67039	4500.379045	4531.578471	4504.129157	4512.028891
511000	63875	13514.61358	4454.559965	4484.92336	4458.063402	4465.848909
512000	64000	13368.39857	4408.070235	4437.770004	4410.537312	4418.792517

Total Messages	Payload Size (kiB)	Input Rate (bytes/sec)	Output A Rate (bytes/sec)	Output B Rate (bytes/sec)	Std. Deviation (bytes/sec)	Delta Rate Input / Output (bytes/sec)
513000	64125	13387.84719	4410.506032	4440.489171	4412.635822	4421.210342