Multi-core architecture for anonymous Internet streaming

Quinten Stokkink

Delft University of Technology

Multi-core architecture for anonymous Internet streaming

Master's Thesis in Computer Science

Parallel and Distributed Systems group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Quinten Stokkink

30th December 2016

Author
Quinten Stokkink

Title
Multi-core architecture for anonymous Internet streaming

MSc presentation
TODO GRADUATION DATE

Graduation Committee

Graduation professor:	Prof. dr. ir. D. H. J. Epema	Delft University of Technology
Supervisor:	Dr. ir. J. A. Pouwelse	Delft University of Technology
Committee member:	Dr. Z. Erkin	Delft University of Technology

Abstract

There are two key components for high throughput distributed anonymizing applications. The first key component is overhead due to message complexity of the utilized algorithms. The second key component is an nonscalable architecture to deal with this high throughput. These issues are compounded by the need for anonymization. Using a state of the art serialization technology has been shown to increase performance, in terms of CPU utilization, by 80%. This is due to the compression (byte stuffing) used by this technology. It also decreased lines of code in the Tribler project by roughly 2000 lines. Single-core architectures are shown to be optimizable by performing a minimum s,t-cut on the data flows within the original architecture: between the entry point and the most costly CPU-utilizing component as derived from profiling the application. This method is used on the Tribler technology to create a multi-core architecture. The resulting architecture is shown to be significantly more efficient in terms of consumed CPU for the delivered file download speed.

Preface

Today, we have a world in which everyone has grown together through the Internet. Multimedia is being shared like never before, greatly influencing the overall information available to the public. For instance, Wikipedia sharing community moderated articles and media. At the same time we also see a crack-down on media with more questionable contents. One can think of Wikileaks data being blocked or shadowbanning and removal of content of certain users on social media websites. Constructive discussion is only possible when all parties have all information (and different opinions). It is therefore vital that information remains accessible. Since the leading anonymizing technology (the Tor project) is really not too fond of hosting large media files or bundles, we will look at a technology which does support this: Tribler. Now, since Tribler uses a Tor-like protocol, which is not suited to host these files for performance reasons (which is why users shy away from Tor), the scientific mission is to investigate the architectural shortcomings of the system.

The author would like to thank Dr. Johan Pouwelse for his invaluable feedback and suggestions during the entire process of writing this thesis. Furthermore, the author would like to thank Dr. Zekeriya Erkin for his assistance during the research phase of this thesis project. Lastly, the author's thanks go out to the Tribler team for providing feedback on the design during the implementation phase of the project.

Quinten Stokkink

Delft, The Netherlands
30th December 2016

Contents

Chapter 1

Introduction

We have come a long way since RFC1[1], which still required the definition of messages as data traveling from host to host. We have passed the early days of the consumer internet where webpages were mostly serving text and images[2]. It would be fair to say we are now part of the digital media age, in which streaming media is the dominant factor in the load distribution of the internet. To name a few of the services contributing to this media revolution, we now see streaming video platforms such as Youtube (over 300 hours of video served per minute in 2015[3]) and Twitch (over 7500 hours of video served per minute in 2015[4]). By estimation the amount of data being transferred over mobile networks in the United States alone is 18 terabytes per minute [5].

At the same time, we also see an increase in the number of reports of censorship by governments and large companies, moderating the content being shared on-line. For instance, censorship of WikiLeaks documents[6, 7, 8] or Facebook and Twitter censoring journalists[9, 10, 11, 12]. Of course, a case can and has been made that these companies are under immense stress to perform this censorship by all sorts of other actors[13]. However, this leaves one searching for a platform providing freedom of expression and freedom of information. One could say that the very core of scientific discourse is under fire.

At the time of writing there exist a number of anonymization technologies, with the Tor technology as the de facto standard[14, 15]. However, the Tor technology is frequently quoted as being suboptimal or barely acceptable in performance[15]. This issue is compounded by the aforementioned transition to more streaming-media centric internet content. This makes the Tor technology particularly unfit for the current climate. Clearly, there needs to be a system which can handle these immense throughputs of data.

One promising adaptation of the Tor protocol is called Tribler[16], which has a Tor-like anonymization protocol implemented[17]. However, this Tor-like protocol is implemented on top of the BitTorrent client, making it more suitable for large files. Still, download speeds are not exceeding 2 MBps with minimal security[17]. One could say that there is still room for improvement.

1.1 Contribution

This thesis will focus on the transition of a single-core architecture to a multi-core architecture in a scalable, performant and maintainable fashion for anonymous file streaming applications. Since the amount of different kinds of such technologies is low and the cost of re-engineering is high, this thesis will utilize qualitative research.

In particular, while presenting several generalizable constructs and algorithms, methods will be implemented and tested with the Tribler[16] technology. This serves the purposes of an example implementation, as well as giving a tangible result for the reader to judge the suggested methods by.

Tribler has been chosen because it offers a significantly hostile environment to multi-core architectures. This is because it has been implemented in Python, which does not allow for two threads to execute code concurrently. This hostility should, in turn, make the results and methods of this thesis more generally applicable, as better multi-threading support can only further the performance benefits for other technologies.

1.2 Research question

The main research question for this thesis is the following:

Where in the architecture of a single-core anonymous-streaming application does one start the multi-core adaptation for the best balance of scalability, performance and maintainability?

Answering this question will involve determining:

1. The architectural requirements of an anonymous-streaming application
2. The parallelizable components of the source code
3. The benefits and detriments of parallelization for particular components

Note that the steps in answering the research question are actually generally applicable for the re-engineering of any application. The answers which will be found for this context will however differ from other distributed applications, through the nature of anonymous-streaming software.

1.3 Problem Description

Anonymizing file-streaming technologies intrinsically have two problematic characteristics:

1. Large data flow volume
2. CPU-heavy channel encryption[18]

The large volume of data is a problem for multi-core architectures, as the multi-core implementations in threads or processes often require pass-by-value communication. In turn, this (especially in a multi-core hostile environment such as Python) pass-by-value interaction requires very high performance data streams.

To complicate matters further, these high performance data streams also need to transport control messages for the anonymization layer. These messages are numerous and diverse in structure, numbering 14 control messages in the original Tor specification[19] and at the time of writing 22 in Tribler. Some form of serialization is required to distinguish the different messages being transferred. This message polymorphism is in stark contrast to the data stream requirement of monomorphism for high performance processing.

On the other hand, the CPU-heavy channel encryption also provides a clear pointer to the components of the application which require a multi-core implementation. Implementing a multi-core architecture for the encryption layer would provide a clear benefit. However, a true parallel implementation of the encryption layer in the Tor stack is still an open problem[15].

To make matters worse, this thesis also imposes the restriction of the architecture having to be maintainable aside from being performant and scalable. This means that the architecture should be as flexible as possible, allowing for implementation changes within the different layers. In other words, it should make use of the existing separation of concerns in the architecture to allow for future modification. In yet other words, the multi-core architecture should be non-intrusive in the existing functionality (as much as possible).

1.4 Document Structure

The rest of this thesis will be structure as follows. In chapter 2 a high-level preliminary introduction to the architecture of anonymizing file streaming applications will be given. In chapter 3 pioneering research on data serialization for interactions in high throughput multi-core or distributed contexts is presented. This is not neccessarily part of, but and important addition to, the actual multi-core architecture creation method which will be presented in chapter 4. Following the method, the actual implementation in the Tribler platform will be presented in chapter 5, followed by the experiments and their results in chapter 6. Finally this thesis will end with the conclusion in chapter 7.

Chapter 2

Anonymous File Streaming Architecture

Whereas chapter 1 focused more on the abstract issues of anonymizing file-streaming applications, it is important to start the initiation of architectural change with gaining insight into an actual implementation[20, 21, 22, 23, 24]. This process of documenting and learning the existing architecture is also known as (software) *Reengineering*, *Software Reverse Engineering (SRE)* or even just *refactoring*. As mentioned before, the corpus of examined platforms for this thesis exists of just one technology: Tribler[16]. This will limit the generalizability of the architectural findings.

To facilitate the understanding of the Tribler project, the two classic reengineering tools will be used: documentation and code inspection. The following sections will serve to document the existing Tribler architecture based on documentation, papers, code inspection and hand-crafted inspection tools.

The following sections will detail the high-level architecture of an anonymous file streaming application and its components. Components which are unique to the Tribler technology will be pointed out explicitly.

2.1 High-level architecture

Generally speaking, an anonymizing file-streaming application exists of three components. The first component is the *peer discovery* component. This component is responsible for finding other users of the applications (also known as peers) which can facilitate the retrieval of files. The second component is the *file streaming* layer. This layer is responsible for managing the files in the operating system the application is running on. Furthermore, it is responsible for defining the protocol(s) to be used between peers. The third component is the *anonymization technology*. This third component makes sure that any data being sent over the Internet is cryptographically secured.

Note that all of these components can be decoupled. One can discover peers

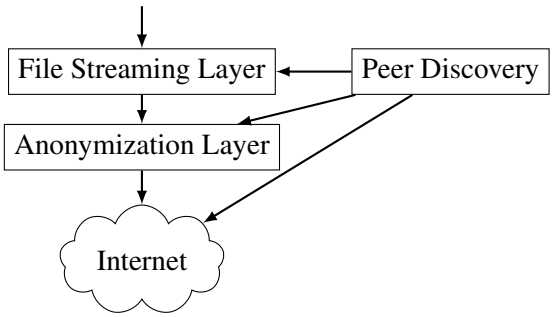


Figure 2.1: Anonymous File Streaming architecture

without sending files or encrypting the communication. One can send/receive files to/from a peer without content centric peer discovery or communication encryption. Lastly, one can encrypt communication, without this pertaining to files or peer discovery. All of these behaviors have been observed both decoupled and in tandem, from code inspection in the Tribler project.

An overview of the interactions between the components of an anonymous file streaming application can be seen in Figure 2.1. Typically, the application initiates the downloading of a file with the File Streaming Layer. The File Streaming Layer then queries the Peer Discovery component for peers capable of sharing the particular resource. Once a capable peer has been discovered, the data exchange control messages and the requested resource can be sent through the anonymization layer.

The next sections will detail the inner workings of each of the mentioned components both in general and in Tribler. This is done because Tribler offers more functionality than a generic anonymous file streaming application, as it also has a decentralized architecture.

2.2 Peer Discovery

Generally speaking in a peer-to-peer network, peer discovery is aimed at finding other users of some application to improve the quality of service of the originating user. In practice, this means that this is a *focused* effort to find peers which share a common interest for some resouce. For instance, in the BitTorrent protocol, peer discovery will be aimed at finding other users which are interested in a specific torrent file (communicated by means of a hash of the torrent contents). This is usually done by means of Distributed Hash Tables (DHT) or Peer Exchange (PEX)[25]. Another example is Tor's hidden services, which use DHT to to advertise a peer's resource via multiple introduction points[19]. This allows for a peer's IP-address to remain hidden while still being able to share a certain resource. The Tribler technology uses a combination of the BitTorrent and Tor technologies to provide torrent-based hidden services (ergo sharing by file content instead of name).

The biggest challenge in peer discovery is usually Network Address Translation

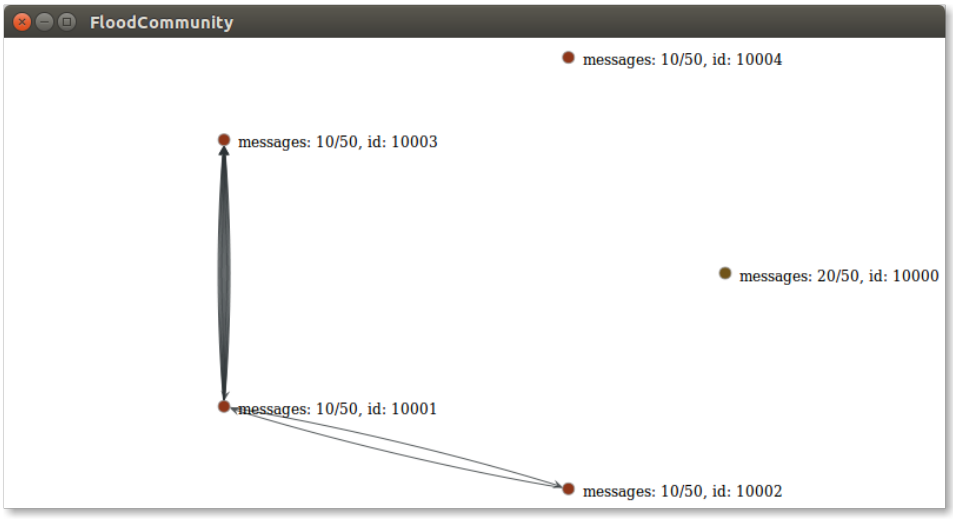


Figure 2.2: Message synchronization and peer exchange in Dispersy

(NAT) traversal. Most technologies use the UDP protocol because it is easier to work with than TCP[26] and has a higher success rate[27]. The same is true for Tribler.

One interesting thing to note about the peer discovery component of Tribler, is that it has two peer discovery mechanisms. Next to the content/torrent based peer discovery, it also features a *community* based peer discovery system called Dispersy[28]. The Dispersy component of Tribler allows for *focused* resource name based discovery (like the hidden services) but with *unfocused* contents. This allows for more message board-like behavior, coupled with being a good starting point for finding peers willing to be a part of a hidden services construction.

Investigation of the Tribler/Dispersy peer and data exchange stack led to creation of synchronization visualization artifacts[1] (see Figure 2.2) and further research into the threading model of the Python backend[29]. The synchronization mechanism of Dispersy and converting the synchronous implementation to an event-driven implementation will remain outside the scope of this thesis however.

2.3 File Streaming Layer

The file streaming layer in an anonymous file streaming application is responsible for file management on the operating system and sharing files (both sending and receiving) from other users. This means that it uses file locks to avoid file corruption while writing and (possibly at the same) time allows for reading.

Several protocols for file exchange (or content delivery) exist, such as BitTorrent, File Transfer Protocol (FTP) and Hypertext Transfer Protocol (HTTP). However,

[1] Available at `https://github.com/qstokkink/VisualDispersy`

the most scalable and fault-tolerant solution to date is the BitTorrent protocol[30, 31, 32]. These are exceptionally important characteristics in a peer-to-peer setting, with peers dropping and popping into existence continuously.

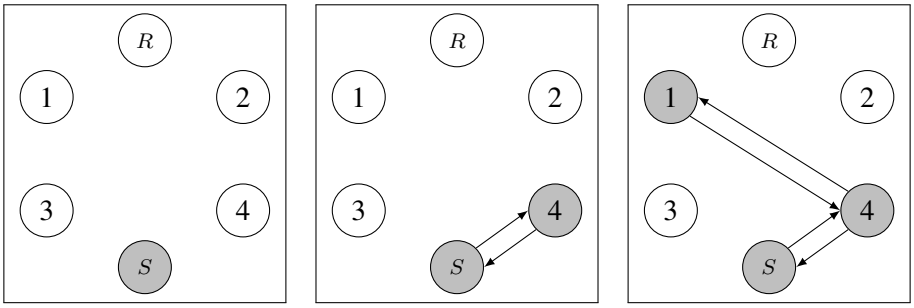
The most important characteristic of this layer, is that all of the implementations named above require a singleton construct to avoid file corruption. This is a very important observation when considering a multi-core architecture.

2.4 Anonymization Layer

The anonymization layer is responsible for encrypting and decrypting data between peers in an anonymizing fashion. This could be implemented by using freenet, GNUNet[33], Tor[19], a mixnet[34] (for example I2P[35] or Java Anon Proxy [36])[14] or some lesser-known technology. These technologies aim to hide the path the data traverses over the internet. Note that this is the most fallible component of the anonymous file streaming application. Many attacks on anonymizing technologies have been known for a long time[37]. At the time of writing, researchers are particularly struggling with traffic correlation attacks[38].

Since the attacks on and the appreciation of anonymizing technology are definitely outside of the scope of this thesis, we shall now proceed to focus on Tor. In Tribler, Tor is also the underlying anonymization technology. To provide a high-level overview of the inner workings of Tor, it can be viewed as a two step process. The first step is to create a path (also known as a circuit) spanning multiple peers between the sending and receiving peer. This first step is the most costly CPU-wise in the Tor protocol and is also known as telescoping[15]. This circuit serves as a means to anonymize the source of the data. The second step is to actually communicate the data through this circuit. There are two things to note about the second step. The first noteworthy thing is that this second step may not happen immediately after creation of the circuit, instead a pool of circuits is maintained. The second noteworthy point is that the circuit does not end at the receiving node, but rather the unencrypted data is sent from the last peer in the circuit (the exit node) to the receiving peer. In Tribler, this exit node is not used when sending from a Tribler peer to a Tribler peer.

To visualize the telescoping procedure, Figure 2.3 shows the creation of a 2-hop circuit (ergo 2 intermediary peers), given 4 available intermediaries and one receiving node R. First the sending node captures node 4. When successful, the sending node will capture node 1 through node 4 (note that there is no direct contact between node 1 and the sending node). In this example node 1 will be used as the exit node. In practice, most nodes will not accept being appointed as an exit node, due to liability issues with the transferred content[39].



(a) 0 hops (b) 1 hop (c) 2 hops

Figure 2.3: Telescoping

Tribler exploits flexibility of the structure of the anonymization layer to allow for flexibility in the file streaming layer. Tribler will send its small BitTorrent packets over all of its available circuits (currently in a round robin fashion) to the receiving peer. In effect, this has become a multipath communication approach. This would even allow for flow control, which is an open problem in Tor[15], though this is not implemented.

Chapter 3

Generic Message Parsing

Whereas most fields in Computer Science are concerned almost exclusively with time complexity, the field of Distributed Systems also has an emphasis on message complexity when designing algorithms. This is a logical result of the early days of the Internet, where transferring any and all data over the Internet was literally an expensive operation. Likewise, the pass-by-value nature of multi-core architectures, as preluded in the previous chapters, shares many of the same concerns. Any of the values shared between components executed in parallel, will require some form of serialization (also known as marshaling). This serialization costs CPU time and will therefore impose the most basic overhead in parallel and/or distributed computation. This is a problem of yore within Distributed Shared Memory (DSM) systems[40]. Many design patterns for synchronizing the serialized objects do exist however[41].

It is important to note that environments which do support shared memory constructs within their multi-core architectures, should use this as it is much faster. However, this is an unrealistic assumption to make for all distributed applications. Applications can and will offload computations to distributed hardware and/or be run in multi-core hostile environments (applications such as Tribler). This makes message serialization one of the core issues in multi-core scalability for distributed systems. This is extra important when dealing with anonymizing (polymorphic messages; serialization needs to be flexible and is therefore slower) file streaming (high volume; serialization needs to fast or the application locks up) applications.

Research on small high throughput communications devices has shown that a 75% decrease in message size led to a 59% decrease in message processing time[42]. Research into online game interactions showed a 84% decrease in message size leading to a 90% decrease in message processing time[43].

3.1 Methods of Serialization

Even though there is some research on the performance of using purpose-built serialization libraries, research on their inner-workings remains largely non-existent. To rectify this, this thesis will lay out the general principles, as they have been inferred from mature codebases. Concretely, this thesis will look at Google's Protocol Buffers[44] and Cap'n Proto[45] to construct a scientific description of an effective serialization protocol. The following subsections will entail the common elements in state-of-the-art serialization frameworks.

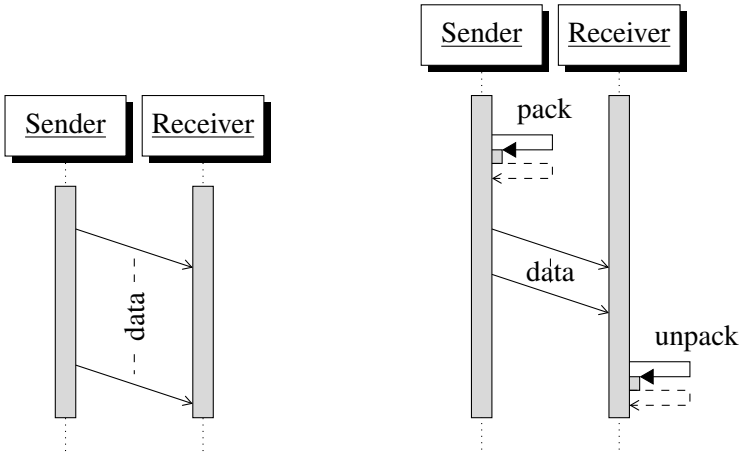
3.1.1 Domain Specific Languages

The first noticeable common element in the design of the aforementioned serialization protocols is their use of a declarative Domain Specific Language (DSL)[46] for the specification of data constructs which can be serialized. This is remarkable, as this requires an additional message parsing structure on top of the serialization mechanism. For the implementation in most programming languages Protocol Buffers even requires the DSL message specification to be compiled before use in an application. In contrast, there exist more implicit specifications based on annotation mechanisms available in the programming language. An example of such an implicit declaration methodology is Java RMI[47].

Another feature which these DSLs support, is *implicit versioning*. Rather than explicitly specifying a communication protocol version to the message type, fields within the message type are assigned a static position by the user. This is used for both forward and backward compatibility. New fields simply get assigned a higher id and deprecated fields are left out in the future. This is another baffling feature, as the rest of the Software Engineering field is moving more and more in the direction of Semantic Versioning[48], ergo even stricter versioning constructs.

One great feature of using DSLs is the built-in type checking. Whereas types would have to be checked explicitly in user serialization and types had to be inferred in annotated schemes, types can now be checked statically when serializing and deserializing. This means that the code which handles the serialization is more generally applicable over all data with the same type (code reuse) and less complex when dealing with runtime types (serializing data *as* some type is easier than serializing *from* some type).

The last characteristic of DSLs is that they are programming language independent. This is a double-edged sword. On one hand, this means that regardless of the programming language being used, the DSL can be integrated. On the other hand, the DSL parser will have to be integrated for every programming language being used. This does allow for easy communication between programming languages though.



(a) No packing (b) With packing

Figure 3.1: Byte stuffing running time

3.1.2 Byte stuffing

One of the key features of the serialization protocols is byte stuffing (also advertised as byte packing). This is a step up from lower level bit stuffing (which has also been shown to improve performance in distributed systems[49]), but still allows for some pretty dramatic downsizing of messages (and an increase in performance)[42, 43].

As has been shown, there exists sparse research on the fact that byte stuffing 'works' for distributed systems. What is still missing is a scientific explanation for *why* and *how* it works. The *how* portion of this question is easily answered: data is packed, sent over the internet and then unpacked, in contrast to simply being sent over the internet. The high-level visualization of the packing process is available in Figure 3.1. Answering the *why* portion of the question is not so trivial though. The following headings will deal will the synchronous and asynchronous cases.

Synchronous case

At first glance one might define the optimization problem as follows. Given some data d which can be reduced to $\tilde{d}$, for packing to have a positive effect on the running time of the entire system, the transfer time of the data $T(d)$ must be larger than the summation of the packing time of the data $P(d)$, the transfer time of the packed data $T(\tilde{d})$ and the unpacking time of the packed data $U(\tilde{d})$. To put this in terms of a speed S of packing versus no packing, we can define:

$$S = \frac{P(d) + T(\tilde{d}) + U(\tilde{d})}{T(d)}$$

Since the transfer time of data can be assumed to be a scalar operation in respect

to the size of the transferred data, we can redefine the speed as follows:

$$S = \frac{P(d) + U(\tilde{d})}{T(d)} + \frac{|\tilde{d}|}{|d|}$$

This means that the packing method will become faster if:

$$\frac{P(d) + U(\tilde{d})}{T(d)} + \frac{|\tilde{d}|}{|d|} \leq 1$$

$$P(d) + U(\tilde{d}) \leq (1 - \frac{|\tilde{d}|}{|d|}) \times T(d)$$

To translate this formula into words: byte stuffing becomes faster than sending raw data when the relative message size reduction is larger than the packing and unpacking overhead relative to the transmission time of the raw data. The implication of this is that, as transferred data gets smaller, the expected message compression quotient $\frac{|\tilde{d}|}{|d|}$ will be closer to 1 and any speedup near unachievable. This means that it only makes sense to use byte stuffing for larger messages (which is conform to intuition).

Asynchronous case

In reality, most implementations will use a socket listener in a thread (*Active Object* pattern), feeding data to another thread to handle it (*Monitor Object* pattern)[50]. Assuming that a sufficient quantity of messages n is sent ($U(\frac{\tilde{d}}{n}) \leq n \times T(d)$, where d is the concatenated data for n messages and $\tilde{d}$ is the concatenation of all packed versions in d), byte stuffing now becomes even more effective, as unpacking can be performed while receiving new messages. In saturated input buffer scenarios we can define the following relative speed:

$$S = \frac{P(\frac{d}{n}) + T(\tilde{d}) + U(\frac{\tilde{d}}{n})}{T(d)}$$

This leads to the following speedup condition:

$$P(\frac{d}{n}) + U(\frac{\tilde{d}}{n}) \leq (1 - \frac{|\tilde{d}|}{|d|}) \times T(d)$$

If the input buffer is mostly in a saturated state, the amount of messages in the input queue n will be very large. This means that the condition is easier satisfied. In other words, in high throughput situations, even with smaller messages sizes, it makes sense to use byte stuffing. This is exactly the case with anonymous file streaming applications.

3.1.3 Real-time serialization

One interesting novelty can be observed from the Cap'n Proto project: real-time serialization. This means that as an object's fields are being written, they are immediately stored in memory in serialized form. In other words, there is no packing overhead when transmitting data to another party.

The downside of this approach is of course that a multitude of changes in an object's fields before serialization leads to additional overhead, compared to serialization on demand. This makes it more suitable for applications which change little and synchronize a lot.

3.2 Tribler's Serialization

Now that we have seen the state-of-the-art serialization methods and their huge performance benefits, it's time to look at a mature anonymous file streaming application. When inspecting the Tribler source code one will find the protocol used for serialization, which is *nothing*. This makes it a good candidate to observe what the source code and behavior of a project will look like if existing technology is ignored.

For starters, one can observe the effect on the code base in terms of project structure. What we have seen is that every synchronizable object is accompanied by a *payload* and *conversion* object. The payload object serves as a data container (also known as a *data class* or simply a *struct*). The payload object also holds routing information for synchronization between peers (for example the signature of the message and whom to confirm this signature with). The conversion object serves to pack and unpack the binary data sent over the internet and instantiate the payload structure.

There are several adverse effects tied to this structure. For one, in the payload definition, there exists a lot of duplicate code and a lot of code which should be duplicated, but is not. One example is the encoding of a 20 character torrent infohash string. Throughout the project there exist 4 different ways to encode the same data. There is a direct 20 character string definition, an unbounded string definition and two different utility classes performing the same function. The same happens for encoding the same *long* value: it can be encoded as a *long long*, a *long*, an *unsigned long* or a *unsigned long long* in different source files.

More or less the same happens in the conversion definition. We have observed the same type checks over and over. Taking one random file with 9 message definitions we count 25 and 18 instances of the exact same field type check (number and string type checks respectively). Since all of these checks require 2 lines of code, we can say for this one example file at least 86 lines of code are wasted (which is 20% of the file size).

Another interesting observation from the code is that Murphy's law is applicable. If programmers are defining their messages in their programming language instead of a restrictive DSL: everything that can go wrong will go wrong. Every so often

the payload classes are not completely payload classes. These data containers have hidden functionality, which is not apparent and not in line with the rest of the project. This is very bad for the maintainability of the code as new programmers in the project will expect this and therefore not look for this. In turn mistakes will probably be made when implementing new behaviors.

One final note about Tribler's serialization is its synchronization mechanism. To synchronize messages between peers, Tribler stores its messages in binary form in its database. This allows for easy retransmission without any compression or decompression of packets. This also means that there is a huge gain to be had by byte stuffed packets. This retains the original constraint of packets being immediately available for retransmission, while taking less space in the database. As a reminder, subsection 3.1.2 explains how packing and unpacking overhead impacts the performance of the application.

Chapter 4

Performant Anonymous Streaming

So far, in chapter 2, we have seen the overal architecture of an anonymous file streaming application. Then chapter 3 explained what protocols to use and why for message serialization in a multi-core architecture. This chapter will actually explain how and where to create the separation in the single-core architecture to transform it into a multi-core architecture. We will start by quickly reiterating the high-level requirements and their consequences. Then the differences of implementation and design between multi-core hostile and non-hostile environments will be discussed. Next, the the different types of data being transferred through an anonymizing file streaming application will be brought to light. Lastly a method to infer the optimal multi-core architecture for an anonymizing file streaming application will be presented.

4.1 Requirements

Thus far this thesis has shown some characteristics of anonymizing file streaming applications, but the requirements have not been specified yet. This section will deal with the actual specification and explanation of the requirements according to these observed characteristics. The following subsections will entail the different requirements and the rationale for their inclusion in a multi-core anonymizing file streaming application.

4.1.1 Maximize parallelization effectiveness

One of the earliest observations of parallel computing is Amdahl's observation that adding more of it, induces more overhead[51]. This, of course, led to the now famous definition of "Amdahl's Law" and is applicable to both parallel and distributed systems. Naturally overhead in distributed systems is even worse than in parallel systems.

Amdahl's observation is, however, a very important one. If one words it differently it becomes: the more parts of your architecture you transform into a multi-core architecture, the less you will gain. In other words, there is negative exponential decay when implementing a multi-core architecture for more components in an architecture. This means that it is very important that, when you switch a component in your architecture, you should choose the component which has the most impact on your running time.

Identifying the most taxing component in a system can be done by using a CPU profiler. These come in various shapes and sizes, being Operating System bound (like DTrace for UNIX[52] or .NET applications in Windows[53]) or bound to the programming language (like cProfile for Python [54] or the Java HPROF profiler[55]). Previous research by the author of this thesis et al. has already pinpointed the CPU-heavy component in Tribler as being the encryption and decryption calls[18].

4.1.2 Preserve shared-hardware-singleton constructs

One of the biggest issues in database access and distributed data, is recovering from concurrent modification (for example in distributed systems after network outage[56]). This leaves the software in a state where it has to choose which changes to throw away (or in the best case how to merge the changes). Naturally for these applications this is unavoidable and a key feature of these systems. For any applications seeking performance benefits this is a problem which should be avoided like the plague.

The core problem for these hard-to-recover concurrent modifications is specifically concurrent modification to a shared resource. On the Operating System level one may think of this as concurrent access to files (with or without file locks) or the graphics card. On the interpreter level (for languages such as Java or Python) one may think of the shared process id or the process executable. For each of these contexts one requires a locking mechanism or singleton structure for access to these shared resources or face corruption of some sort.

This means that, when switching to a multi-core architecture, one first needs to consider the context of the switch. We can distinguish the following contexts per different method of multi-core implementation:

1. Threads: Interpreter context
2. Processes: Operating System context
3. Distributed: Universal context

These contexts are ordered in increasing concern. In other words when switching an architecture to a threaded multi-core architecture, one should make sure to keep all singleton constructs and locks in tact protecting the state of the interpreter, the

operating system and the universe. When switching an architecture to a multi-process multi-core architecture, one needs to sustain all singletons maintaining the state of the Operating System and the universe. In a multi-process multi-core architecture one can refactor the interpreter singletons to their own process. Lastly one might wonder what a universal singleton is: in this thesis this is defined as a third party maintained shared object. Practical examples would be adhering to the transaction mechanism of a database server or maintaining ones id in a server ring.

4.1.3 Preserve anonymization

There are four kinds of application functionalities which can be exploited by *hackers*: input validation, authorization, race conditions and unexpected interactions[57]. As one might expect, switching to a multi-core architecture opens up all of these exploit paths. Especially when switching to distributed hardware over the internet, the data exchanged between cores is a very big issue.

An example of a possible unexpected interaction is with the circuit creation for the Tor circuits. Normally a peer chooses other peers to create a circuit with. What can happen if multiple instances of the Tor stack are run on the same machine (using threads or processes for the multi-core implementation) is that they use the other local instances for their circuits. So instead of using multiple physical machines, as intended, for anonymization: one loops with oneself, rendering the anonymization technology useless.

An obvious exploit relating to sharing data over the internet is not only sending it encrypted, but making sure it is not altered or duplicated. One easy exploit that comes to mind would be to use a replay attack and thereby denying service or worse yet, screwing up the encryption itself.

Note that the context in which one creates the multi-core also defines the possible attacks. For different use-cases one might consider different levels of protection built into the message passing layers. For example, thread based communication may limit itself to interpreter-level attacks, which are basically non-existent over attacks which would already be possible on the single-core architecture. Multi-process implementations without socket listeners are also not much more unsafe than their single-core counterparts. When one shifts to a multi-process architecture with socket communication or a LAN distributed system, one should start worrying about input validation and authentication. Having a WAN distributed system is of course the most dangerous and opens up all attack vectors in regards to message transformation and interception.

4.1.4 Allow for high throughput

The big caveat of anonymizing file streaming applications is the file streaming component. This means that large amounts of data will be passing through the application. If this data crosses the same paths as control data, it will bring the

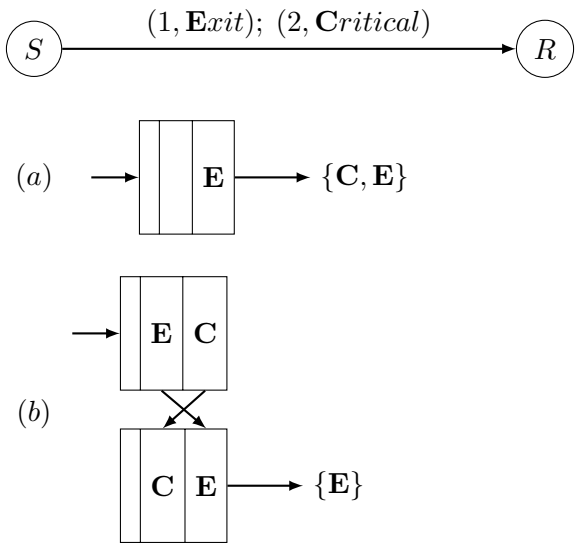


Figure 4.1: Non-determinism in priority queues

application to a crawl. This is usually a big concern, as this hurts the end-user experience of the application[58].

There are two ways to avoid a decrease in application responsiveness. The first way is to prioritize control messages over data messages. The second way is to separate the control message flow from the data message flow. The following headings will deal with both of these solutions. Note that both methods will benefit from fast message serialization and minimal data transfer, but this will remain outside of the scope of this section.

Prioritization

The most difficult option to maintain responsiveness is message prioritization. This means that messages will be assigned tags by the application programmer, by hand. Note that prioritization may happen at the sending or the receiving node. The benefit of this solution is that there is only one channel required between the sender and the receiver of the message.

The easiest structure for prioritization is a sender-only priority send queue, with a FIFO queue for the receiver. This leaves the entire system in a deterministic state, where the sender can guarantee the message ordering as it has been sent. This kind of structure makes sense if the sender has an outbound message queue and the receiver can keep up with the message flow. In practice this will probably be the other way around though, most protocols such as TCP seek to increase the data flow until the receiver can no longer keep up (scaling window)[59].

The more complex solution is to implement a priority queue for both the sender and the receiver. This does introduce non-deterministic message ordering though. This system can be very dangerous when dealing with critical messages. Having

both the sender and receiver use a priority queue is the fastest method for priority based messages though.

An example of this non-determinism is given in Figure 4.1. Suppose a sender S first send a critical message C with a lower priority and then an exit message E with a higher priority, to a receiver R. The first option (a) is that the critical message has left the queue and has been handled already before E enters the queue: this will lead to the execution of both C and E. The second option (b) is that the critical message C is still in the queue when E arrives: the priority queue will switch the two and handle E first. Since E inhibits C from being handled (in this example it signals a hard exit, in practice this may be more complex), we see a different set of statements being executed: only E. In other words, for the same order of sent messages C and E we have two different unpredictable outcomes, this is non-determinism.

If we look at Tribler we see a variation of the more complex solution used, it offers no send queue but the receiver has a priority-based input queue. Also, Tribler uses a hybrid priority *and* FIFO queue. One troubling observation is that the default priority reigns supreme over the messages which do use the priority queue though, making it effectively worthless. Of the 9 messages which use the priority system, only 2 use a non-default priority (this is 22%).

Channel Separation

The easier option to separate control messages from data messages is by assigning them different channels. This can be done by using sockets (on the Operating System level), file descriptors (on the process level) or queue constructs (on the executable level). Note that these options are ordered in increasing processing speed and decreasing applicability. A downside of this approach is that the programmer will have to manage these multiple channels, with the risk of introducing shutdown race conditions and possibly other unexpected channel interactions.

The fastest and therefore most desirable option would be for applications to use double-ended queues to deliver messages. This would mean that messages never leave the *application's memory*. This kind of construct is easier to check for both the sender and receiver and will shut down as the executable is shut down. So aside from being the fastest option, it is also the safest. The downside is, of course, that it is only usable in shared-memory contexts: such as threads or processes with a shared memory construct. Fast implementations for dealing with these kinds of shared memory applications already exist: for instance OpenMP, MPI or Pthreads[60].

Next in line for channel separation would be using file descriptors or pipes between processes. This means that the messages never leave the *machine's memory*. This is a step up from using threads, being slower and less safe. In the worst case, open file descriptors can cause blocked processes (and therefore orphaned processes) and physical artifacts on a machine. Especially on less powerful hardware, machines can potentially run out of file descriptors and/or (extra overhead) memory

to use for processes[61].

Arguably the worst way to provide multi-core architecture support is by using sockets. Although this is unavoidable in distributed hardware. One big issue with using sockets scalably is the sheer number of sockets already in use and the high chance of collision: the number of known ports in use is reported by Wikipedia at 1399 (including collisions) on the range of port number 0 through 49151[62]. Another issue with sockets is that it opens up a path for external communication into one's hardware. Unless access is restricted to a LAN or by a firewall, sockets are a big security issue compared to the other options.

4.2 Multi-core in a hostile environment

Parallelism can be implemented on many different levels. One can parallelize processor instructions into micro instructions and one can build entire distributed systems. This thesis will draw the line at hostile environments, meaning environments where parallelism can only be introduced outside of the application level. This means that there exist no shared memory constructs between processes or parallelism within a process.

One might be tempted to say that it is useless to introduce parallelism on this level, as all or nearly all programming languages support threading or concurrent flows. A major benefit of this hostile environment approach is that it leaves the different processes room to use their own programming language. This makes it easier to optimize applications when refactoring different components to parallel sections. One could for instance decide to implement a time critical portion in C instead of a slower (semi-)interpreted language like Java, Python, Ruby or Scala.

A second reason why it makes sense to use multiple processes instead of parallelism inside of a single process is bad thread management within the process. The rationale for this can range anywhere from confusing multi-thread constructions using the same resources to actual bad thread implementations offered by the programming language. The former is caused by the programmer working on the project, but the latter we can find in a language like Python.

To give an example of this hostility to normal threaded execution we will momentarily zoom in on Python's thread model. Python disallows threads to execute the same block at the same time, to provide thread safety[29]. Now consider the pseudocode given in Algorithm 1. If the thread's id is equal to 0 it will block until some other thread releases the resource it is waiting for. Normally another thread would then run this code and release the resource, allowing the thread with id 1 to continue execution. In Python however, the other thread can't enter this code as the thread with id 1 is already running it, this will cause the other thread to wait until it exits the code block. What this mundane code block has created in Python, is a deadlock.

Algorithm 1 Thread safety deadlock

```
if id = 0 then
    WAIT FOR(resource)
else
    RELEASE(resource)
end if
```

4.3 Practical data streams

To discern different types of data, we can take a look at another Distributed Systems field: Big Data. In particular we will see how applicable the three V's are which are applicable to that field: Variety, Volume and Velocity[63]. It turns out that all of these V's are applicable to an anonymous file streaming application. The next subsections will detail how and why certain messages in an anonymizing file streaming application have these characteristics. Note that these different data types are good candidates to use different channels for, as discussed in section 4.1.4.

4.3.1 Control messages

As presented in section 1.3, Tribler's anonymization requires the definition of 22 unique messages. This set of pluriform messages, belonging to the *Variety* category of the three V's, can be categorised as control messages.

Typically these control messages have a very wide range of contents. They are also usually very limited in size, in Tribler, rarely breaking $1Kb$. From the fact that Tribler idle runs (only control messages) produce between 2 and $4Kbps$, we can also infer that the amount of messages being sent is not that great in number. To sum up this message category, control messages are:

- High Variety
- Low Volume
- Low Velocity

4.3.2 Data messages

The cornerstone of a (anonymizing) file streaming application is streaming files as quick as possible. Usually this is done in fixed size chunks, like it is done in BitTorrent and therefore the Tribler implementation[64]. The transport layer will then take care to maximize the throughput of the network connection, as discussed in section 4.1.4.

This class of messages specializes in transferring raw data *as fast as possible* and therefore sees very little to no variation in the message structure. Because of the application the volume of these messages is huge though (this is also why the

Tor project would rather not route BitTorrent traffic). To summarize this category of messages:

- No Variety
- High Volume
- High Velocity

4.3.3 Exit messages

The last category of messages is actually a single message, which seems like a normal control message. This is the request-for-exit message. This is a special type of control message, which can very much hurt scalability if not processed in a timely fashion. Especially in distributed systems this is a very big problem.

Exit messages typically have little to no variety, optionally with a reason for termination. They are only sent once, and exactly once to the receiving party. Usually exit messages also include a response, containing information about the termination (which can be a clean exit, or an exit with an exception). For application responsiveness it is imperative that exit messages are handled immediately. If a receiver has locked up in its control flow, it should still be able to exit. The same is true for a receiver with a filled control message input buffer, exit messages should still go through. Failure to adhere to this behavior can and will lead to lock-ups of applications, waiting for a long time or never finishing due to buffered or dropped exit messages. To sum up the exit message category, it has:

- Low Variety
- Low Volume
- High Velocity

4.4 Optimal parallelization

As has been mentioned in subsection 4.1.1, one cannot parallelize effectively without knowing the most taxing component in the architecture (which can be parallelized), in terms of CPU running time. Also, as mentioned in subsection 4.1.2, not all parts of the original architecture can be parallelized. With these restrictions in mind, we must now attempt to create the most effective separation of application components to create a multi-core architecture from a single-core architecture.

To create a separation, one must first obtain a graph of all components in the application and their interactions. Note that this is not necessarily a class diagram. This method is applicable for any granularity of component separation. One could go as fine grained as create a graph of instruction blocks, all the way up to groups of packages and anything in between or a combination. The only restriction is

that the components must be separable. That said, some parts of an application will most likely be more convenient if kept together, from a Software Engineering perspective. For instance, it could hurt the maintainability of an application if a functional component is split apart.

Once one has created a graph containing all of the components (as discussed) in the application, it should be weighted. This weight should be the message flow of the data being transferred between the components. This is a lot of work and therefore imposes a course granularity of the component graph, from a practical viewpoint. First the message sizes must be constructed, which entails the size of the parameters a component is called with when serialized into a message struct. In some cases the message size is not static and will have to be determined as the average message size. The message flow can then be calculated as the average message size times the average amount of calls per second. When determined, all components should be connected by one or more weighted edges, expressed as bytes per second.

As the reader may guess by now, the graph will be optimized for a minimum data flow between groups of components. The rationale for using data flow, is to measure parallelization overhead. The CPU runtime improvement is already guaranteed by offloading the functionality to other cores. In fact, the only parallelization overhead in a running system (where the set-up has already been completed) is the data communication overhead. The more data needs to be communicated, the slower the application becomes. One could in fact see this as a measure of component autonomy which goes beyond conventional class or package coupling.

Once the weighted graph has been constructed for the application's single-core architecture, one can optimize the parallelization. This can be done by constructing a minimum cut in the graph (note that the minimum k-cut with varying k is known to be NP-complete[65]). There is a small twist to the problem, pertaining to the fact that the optimization target may not be in the same partition as the main control flow of the program (this is also known as a *minimum s-t cut* problem or generally a *max-flow min-cut* problem, which can be solved using the Ford-Fulkerson algorithm[66] or derivatives[67]). The partition of the graph including the parallelizable target in the application, can then be split off from the single-core architecture to form the multi-core architecture. Recall that this can be offloaded to a thread, child-process or remote process.

To end on a practical note, as noted before, it makes sense to use a coarse granularity for component distinction. However, it also makes sense to use a more fine granularity for the components near the structure of the optimization target. This is because these components are more likely to be a candidate for an efficient split in architecture.

Chapter 5

Implementations

In chapter 3 this thesis has presented the underappreciated technique of message serialization. Then in chapter 4 requirements, restrictions and a method of creating a multi-core architecture from a single-core architecture has been presented. To provide isolated results, these methods have been implemented separately. This implementation isolation makes sure there is a causal link between the method and the performance changes.

As mentioned earlier, these implementations will be created for the Tribler technology. In section 5.1 the message serialization implementation will be discussed. Then in section 5.2 the multi-core architecture creation and implementation will be discussed.

5.1 Message Serialization

As discussed previously, Tribler uses Dispersy for its communication between users[28]. So it is on top of this component of Tribler that the message serialization will be implemented. To perform this implementation this thesis will first discuss the Dispersy infrastructure and the method of serialization used. Then the new structure, after implementation of this serialization, will be discussed.

5.1.1 Dispersy

To quote the Dispersy GitHub page, Dispersy is an "*elastic database system*". In practice it is more akin to a message propagation protocol, which allows a per-message network propagation definition. Furthermore, it allows for message scoping within a certain per-peer interest context, it calls these contexts *communities*.

The peer discovery component of Dispersy works by peer gossiping using introductions to other peers, based on community. Once a peer has been introduced and accepted in a community, messages pertaining to this community can be communicated between peers. This is a slow, but scalable way to share messages in large communities: local views of members of this community will intersect and this

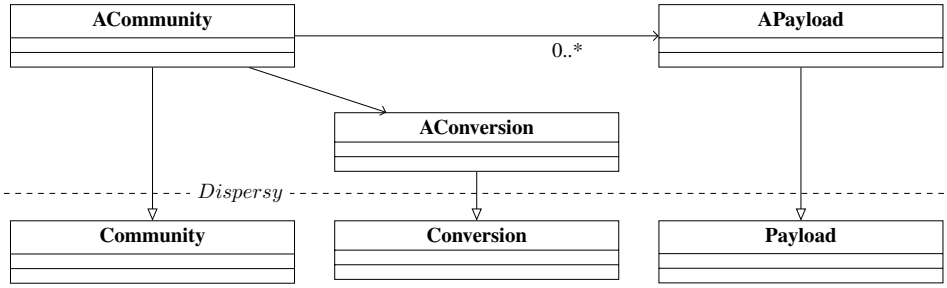


Figure 5.1: Implementing a user community in Dispersy

causes messages to be shared between local clusters of these community members. To facilitate this behavior messages consist of five key components:

- Authentication
- Destination
- Distribution
- Resolution
- Payload

The authentication portion of the message defines which member(s), if any, authorized this message. The resolution portion of the message defines how to confirm the authentication signature. The distribution portion then defines how the message should propagate through the network (whether it should be shared with everyone, a group of members or just one other member). The destination then specifies which members to send the message to. Finally the most important part for the functionality and for this thesis, is the message payload. To maintain compatibility with the rest of Dispersy, only the message payload will be serialized.

A diagram of the resulting architecture of implementing a user community with Dispersy is given in Figure 5.1. Here a user community *ACommunity* is implemented. This community requires the implementation of an *APayload*-type class for each message and an entry in *AConversion* for each of the *APayload*-type classes.

5.1.2 Serialization method

As mentioned before the existing Tribler serialization method is direct serialization using the Python `struct` library. This library allows programmers to serialize data by providing a type, or a sequence of types, to serialize given data to. For instance, one could define a message format M as follows:

$$M = !20shl$$

Technology	Format Read.	Serialized Read.	Size	Avail.
XML	G	VG	VB	S
JSON	G	G	M	S
struct	B	B	M	S
Cap'n Proto	VG	VB	VG	C
Protocol Buffers	VG	VB	G	T

Table 5.1: Serialization technology comparison

This means that (in order of symbols) a big-endian, 20 character string followed by a signed short and a signed long will be serialized (capitalization matters). Note that this is not very much adapted to human-readability.

To select a serialization method to replace this with, several methods have been examined. The selection criteria included:

- Format readability
- Serialized readability:
- Serialized size
- Availability

The format readability is a measure for how well a human can understand the message formatting. This is useful for the **design/modeling** of the messages. The serialized readability is a measure for how well a human can understand the serialized message. This is useful for **debugging** applications, when raw i/o is inspected (optionally using some debugger which allows memory inspection). The serialized size concerns the size increase or decrease of a message's serialized form compared to the size of the original data. Lastly, the availability concerns whether or not the package available in the standard Python distribution, multi-platform third party distribution platforms or if it requires a custom build. The availability is score on a scale of $[\mathbf{S}tandard, \mathbf{T}hird\ party, \mathbf{C}ustom\ build]$ respectively. The other selection criteria have been scored on a scale of $[\mathbf{V}ery\ \mathbf{B}ad, \mathbf{B}ad, \mathbf{M}ediocre, \mathbf{G}ood, \mathbf{V}ery\ \mathbf{G}ood]$ and the results of this selection can be found in Table 5.1. Note that this table is the outcome of discussion between a handful of people, future work is to have the presented technologies judged by a body of people of more significance.

Since this thesis has determined that serialization size is a very important factor (see chapter 3), Cap'n Proto is the obvious choice. However, this choice was also rejected by the Tribler team. The reason for this is the fact that Cap'n Proto is not available in either the standard Python distribution, or third party distribution platforms. So to ease Software Engineering concerns, the Protocol Buffers technology was chosen.

5.1.3 Message design

After the underlying technology has been chosen, the message design can proceed to happen. This process involved extracting all of the 53 old messages format definitions in Tribler (defined in the *struct* format) and porting them to the Protocol Buffers DSL. This also exposed a lot of inconsistencies in serialization of the same data in different messages, as pointed out in section 3.2. Due to the loss of the old *conversion* and *payload* structures, a little over 2000 lines of code were lost (which is about 1% of the $174,000$-line Tribler source code).

During this message porting process another important concern surfaced, which was for the implementation to be backward compatible with the old wire format. To alleviate this a boilerplate layer was constructed on top of the Protocol Buffers serialization. In turn this also allows for the serialization implementation to change at a later date (should Cap'n Proto become more interesting to the Tribler project at a later date, for instance). What the boilerplate layer allowed was to receive messages from both the old and the new formats, while leaving the application the choice in which format to send data. This still leaves a problem open though: when do the the peers start using the new wire format with each other.

To illustrate the protocol problem, we shall define the **Spies in an Enemy Bar problem**:

> *American spies gather in a German bar, where no Germans speak English. Only the master spy knows the identities of all other spies. If any of them start talking in English while there are still Germans around, they will be shot. They want to talk English, but no American spy wants to be shot.*
>
> When do the American spies start speaking English?

The solution to this problem is: when another person in the bar starts talking English. Since Germans can't speak English, if another person is speaking English, they must be a spy. Since no spies want to be shot, they must know there are no Germans in the room. If there are no Germans in the room, it is safe to start speaking English.

This problem is not just for fun, but can be used in practice to design message backward compatibility. In a system, all peers start by sending messages in the old wire format. Once a peer received a message in the new wire format, it will also start sending its messages in the new format. For, when a peer receives a new format message, it must mean that someone knows it to be safe for everyone to start using the new message format. This safety will be assured by the application owner's deprecation labeling of the old wire format, once enough users switch to a compatible version.

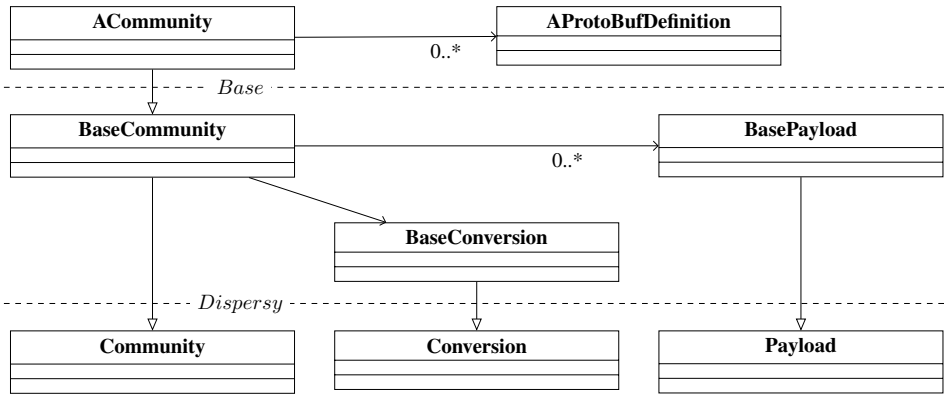


Figure 5.2: Implementing a user community with a BaseCommunity

5.1.4 Resulting Architecture

The resulting architecture of this serialization design is a single Dispersy community structure, dubbed the *BaseCommunity*, from which all other communities can inherit. To retain compatibility with the old design, the new messages are serialized into the payload of a Dispersy message. This led to the BaseCommunity only sharing one type of message, the *BaseMessage*, which had the serialized version of the old messages as its payload. Contrary to the old system, which statically defined the authentication, destination, distribution, resolution and payload, this new message dynamically adapts to its serialized contents. This allows for more flexible message definitions, while retaining support for the old static definitions.

All of the existing Tribler communities had to be rewritten to inherit from the new BaseCommunity. This changed nothing in respect to received message handling within the communities, as the old conversion and payload were kept for backward compatibility using the boilerplate layer. What did change within the existing communities is the switch based sending of messages to other peers, which should be removed in later versions.

The resulting architecture can be observed in Figure 5.2. Instead of the huge code backlog imposed by Dispersy's Payload and Conversion classes, the only definitions a user community *ACommunity* has to deal with now are its *AProtoBufDefinition* definitions. These are much smaller in size and much easier to read (as established earlier in this chapter).

5.2 Multi-core branch-off

As mentioned repeatedly, the optimization target in the Tribler software was the encryption. This was the starting point for employing the architecture optimization and to start growing the parallel component in the new system. In this section the actual layout and interactions of the existing components in Tribler will be discussed. Then the implementation complications will be discussed, followed by

the resulting architecture.

From a code base management perspective the multi-core architecture was decided to map onto processes. The rationale was, as mentioned in section 4.2, that Python is too hostile for a threading implementation. Furthermore, a Tribler user is not expected to own multiple machines. This leaves processes as the most performant choice.

5.2.1 Existing architecture

Following the suggestion given in section 4.4 the existing architecture will be modeled in an increasingly coarse grained fashion starting from the encryption. For ease of explanation, this subsection will actually explain the different components the other way around (starting with the most coarse granularity). In the most high-level view (see Figure 5.3), Tribler consists of two components:

1. Communities

2. Session

The communities component of Tribler (mostly) passively collects and shares Dispersy messages. The Session component of Tribler manages all of the torrent downloads. These are the two main interfaces Tribler uses to communicate with the outside world.

There is one Dispersy community in the communities component responsible for all encryption and peer discovery. This community is called the TunnelCommunity. To be more precise, the actual community handling all of the encryption is the HiddenTunnelCommunity (HiddenCommunity for brevity), which inherits from the TunnelCommunity. What this HiddenCommunity offers over the TunnelCommunity is a Tor hidden services-like implementation on top of the Tor-like communication offered by the TunnelCommunity. This allows for hidden seeding (sharing) of torrent files, instead of just hidden leeching (downloading) of torrent files.

In the Session component there exists a torrenting implementation (provided by the `libtorrent` library). Because this torrenting implementation can only communicate via socket, the data flow has to be brought back to the application from the socket communication if the Tribler user wishes to use encryption. To perform this the communication is forwarded to a local port, where a custom SOCKS5 local server implementation is listening for connections. This SOCKS5 server then forwards the data to the TunnelCommunity, which uses the usual Tor-like protocol to forward this data through circuits.

Finally, note that this architecture description does, by no means, cover all of the interactions in the Tribler software. However, this description is sufficient to form an understanding of the encryption in Tribler.

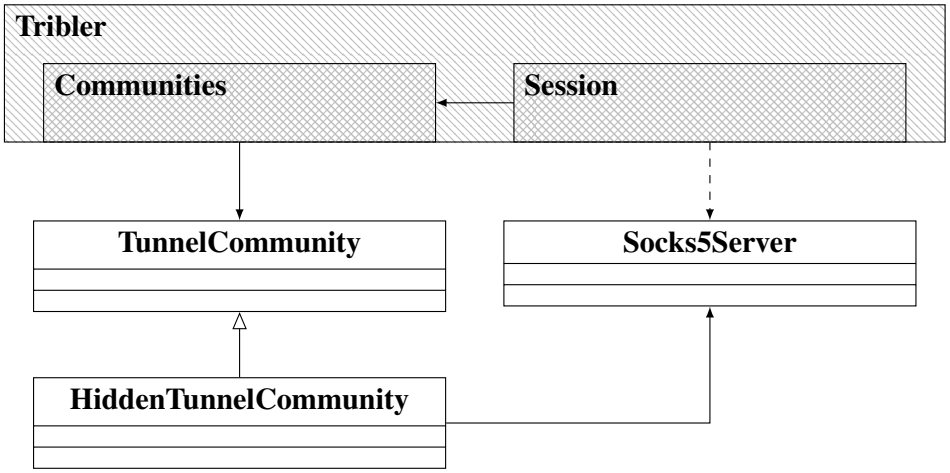


Figure 5.3: TunnelCommunity-centric architecture overview

5.2.2 Soft coupling

As one can see from the architecture overview from last subsection, there is a dependency between the Session component (most importantly the Session class). The Session class is in fact a god class within Tribler, being used by all of the communities for execution.

This singleton dependency is a problem for the implementation. As mentioned in subsection 4.1.2, Operating System context singletons cannot be included when parallelizing to processes. This would trivialize the ability to cut the architecture, limiting it to just the encryption and decryption functions.

Thankfully, even though the Session has a very strong coupling, it can be re-instantiated for other processes in case of the TunnelCommunity. This is done by giving each process running a TunnelCommunity instance its own (severely inhibited) Session singleton, with its own folder in the Operating System folder to perform file I/O in. The only required shared construct with the original Session is the private key of the community member, which is already stored on disk. Disk storage is arguably very unsafe, but this is outside of the scope of this paper.

5.2.3 Creating children

When creating processes from a process, one has two choices. One can either *fork* or *spawn* a process. Forking involves providing a reference of the parent process's memory to the child process, which then has `copy on write` access. This means that it will *read* the parent process's memory, until it needs to change a block of memory. Then it is assigned its own block of memory. Spawning a process, on the other hand, involves giving a child process a completely blank memory allocation. The next part of this section will discuss some of the benefits and downsides of forking and spawning.

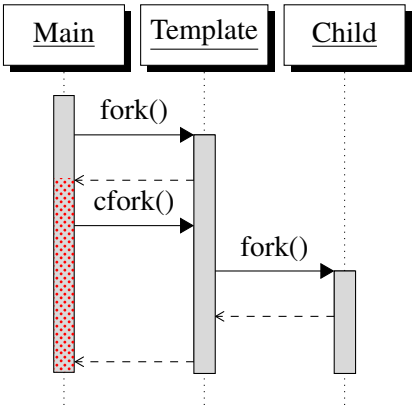


Figure 5.4: Optimal forking architecture

Forking

The big benefit of creating child processes by using forking is reduced Random Access Memory (RAM) usage. If at all possible, this should be used over spawning a process. There are two big caveats to using forking though. The first caveat is that the forked process inherits the memory from the parent exactly as it was before forking. The second caveat is that the inherited memory may include read-only singletons which no longer apply to the child process.

The inherited memory snapshot can be a problem, as Python cannot garbage collect unloaded modules very well. Since any module that offers non-trivial functionality will update itself periodically, this will lead to module copies in child processes which make no sense. Avoiding this can be done by determining the shared libraries between the child and the parent processes. A fork can then be made when these shared libraries have been loaded and this fork can then be forked to create the actual child processes. This allows for optimal memory usage, but also adds a level of indirection to the architecture. An overview of the optimal forking architecture functionality can be found in Figure 5.4: as the main process dirties its memory, the template still has the reference to the original clean memory.

The inherited memory can also contain process specific singletons. One can think of things like the process id or thread manager singletons. Examples include the `twisted` thread manager thinking it was running before the child process started it and threads being managed which the child process did not own. This unexpected behavior can all be explained as being due to inherited self reflection of a process. This is the second reason why Figure 5.4 is set up the way it is.

This tandem of severely dangerous caveats is what ultimately led to using process spawning instead of forking. The risk of very-hard-to-trace bugs is simply too great.

Spawning

Using spawning to create a process does not offer any benefit, other than the assurance that the code being run does not run into *dirty* inherited artifacts. The price for this is heavy in memory usage. The Python interpreter takes 5 MB without any additional loaded packages, when forking this would actually be 0 MB. Loading Tribler takes about 25 MB when spawning, but only 16 MB when forking. In other words, one buys program correctness for scalability in less powerful devices. A typical 8 GB RAM machine will be dropped down from 500 instances of Tribler to only 320 instances. This is a 36% decrease in the maximum amount of runnable processes.

Another disadvantage of spawning is that it requires self reflection on the executable of the application. This is problematic as Tribler can be run in three different ways:

1. Runnable script
2. Python interpreter with script argument
3. Compiled into executable

This means that for scalable code, every distribution will have to be hard-coded into the source, to run a different child executable. This is because the command line invocation of each of these options is different, with different execution paths, different executable paths, different environments and so on. This greatly complicates finding the child executable.

There is however one (and only one) multi-platform way to spawn child processes: by not having a child executable. What the main executable needs is a switch for execution as the normal application or execution as the child application. This switch is controlled by the only safe multi-platform available variable: a command line argument. All of the complicated platform specific things like executable, environment, etc. can then simply be copied from the parent executable.

Finally, note that when spawning a process there is no longer any indirection needed, as it is when forking. Since the child's memory is clean anyway, the main process can simply invoke a spawn, regardless of the state of its own memory.

5.2.4 Optimized architecture cut

When performing the architectural cut in Tribler, the project team's requirements were clear. There would be no cutting in the any of the Tribler functionality which does not concern the encryption and/or decryption. Furthermore the TunnelCommunity itself should remain unscathed. This was very valuable input for constructing the component graph of Tribler, to perform the optimal architecture cut on. It allows for making a much more manageable component graph. Do note that a key observation in making this possible at all was the soft coupling of the Session class.

In fact, the component graph is equal to the one presented in Figure 5.3. Because there is only cutting edge between the TunnelCommunity and the Tribler communities component, this is always part of the optimal cut. The only cutting choice to be made for an optimal cut, was to either include the *Socks5Server* class in the parallelization or not. In other words, is the cut made between the *Session* component and the *Socks5Server* or is the cut performed between the *Socks5Server* and the *HiddenCommunity*. To adhere to the cutting method we have to find out which of the data streams is the largest, for the optimal cut. In this case we can derive the inequality for the message stream sizes without attaching profilers to the application, which saves time. To do this we must inspect the source code of the *Socks5Server*, which we will spend the remainder of this subsection on.

The Socks5Server class

The *Socks5Server* class is a local listen server using the SOCKS5 protocol[68]. It accepts UDP connections, which originate from the `libtorrent` library used by the *Session* component. What the class accepts are `CONNECT`, `BIND` and `ASSOCIATE` requests. When UDP datagrams are received by the *Socks5Server* it will choose a circuit from the *TunnelCommunity* and then call the `tunnel_data()` method on the chosen *Circuit* object. When a response is sent over the circuit it will be delivered by the *TunnelCommunity* to the *Socks5Server* through the `on_incoming_from_tunnel()` method. This will, in turn, be forwarded back to the `libtorrent` library in the *Session* component.

Viewing these interactions abstractly, there is a strictly larger message flow between the *Session* component and the *Socks5Server*, than there is between the *Socks5Server* and the *TunnelCommunity*. The data flow between the former pair consists of both UDP connection messages and UDP wrapped data payload, the flow between the latter just consists of the data payload. Thusly, without measuring the actual flow, we can conclude that the flow between the *Socks5Server* and the *TunnelCommunity* is smaller than the message flow between the *Session* component and the *Socks5Server*. This means that the optimal cut is between the *Socks5Server* and the *TunnelCommunity*. The final architectural cut is shown in Figure 5.5.

5.2.5 Dynamic worker pool

To allow for scaling behavior and to switch between testing forking and spawning (results discussed in subsection 5.2.3) a process manager has been implemented. This process manager follows the worker pool pattern (also known as the master-worker pattern), as this is know to allow for fine tuning in parallelization[69]. As far as the data flow is concerned this is merely a decorator of the flow in the architectural cut.

This is a very practical and important bit of in-between code though. It allows for transparent fault recovery and process management. Furthermore, this can define

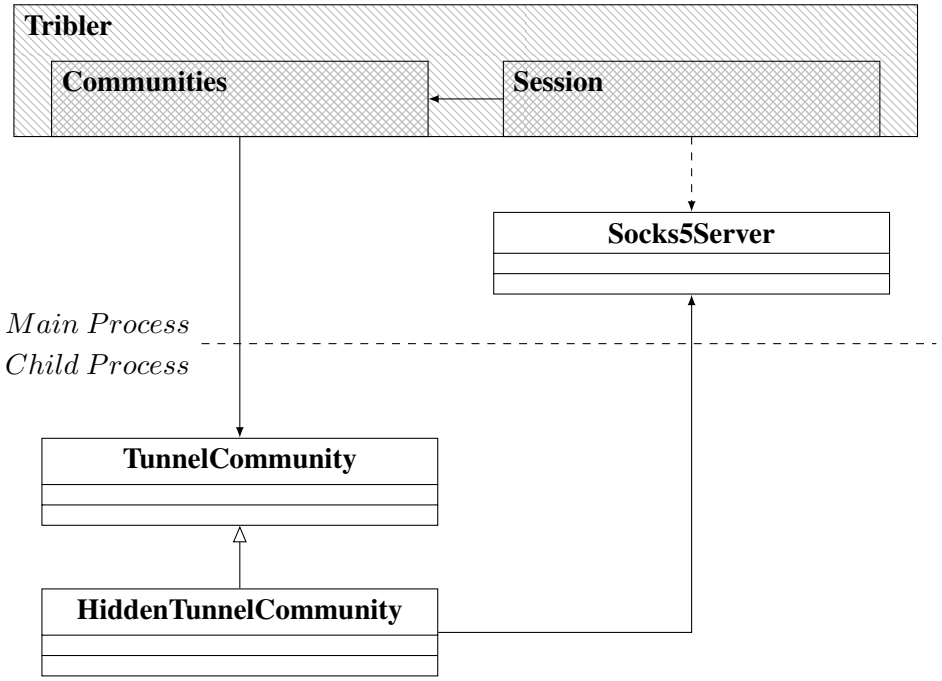


Figure 5.5: TunnelCommunity-centric architecture cut

scaling behavior according to Operating System based measures (such as number of cores and current utilization) and programmer inputs (such as required amount of Tor-like circuits).

Lastly, by implementing this abstraction for the main process, it allows for the multi-core components to change. What this means is that part of the architecture could be offloaded to distributed child processes instead of (just) local processes. This is a way of both future proofing the application and improving maintainability and extensibility.

5.2.6 Message passing interfaces

As mentioned in section 4.3, all of the communication between the main process and each of the child processes should flow through 3 streams: the control, the data and the exit streams. These flows are bidirectional by conception, but have been implemented unidirectional for simplicity of the line protocol and to conform to the unidirectionality of the `stdin`, `stdout`, `stderr` standard streams. The remainder of this sections will describe the low level message passing API as viewed from the main process and a child process. Also, the shared remote procedure call construct will be discussed.

Low-level Views

From the main process's point of view, a pool of *ChildProcess* objects is managed. Such a child process has a process id (assigned by the Operating System) and read and write capabilities to the std, control, data and exit streams. Creation of child processes is handled by the `twisted` library, which is multi-platform and has built in support for custom file descriptors. Child processes can be requested to exit and force-exited if necessary from this view.

From the child process's point of view there only exists one process, the main process. The child process is responsible for exiting itself when requested. Because there is no (at the time of writing) framework or library for listening on file descriptor streams, this had to be implemented from scratch for the child processes.

Generic RPCs

As identified early on, control messages are almost always in the form of a remote procedure call (RPC). Which is to say that they request execution of a remote method and expect a response. This led to a standardized RPC abstract class to separate concerns in the design. This also allows for better testability of the framework.

The RPC calls are implemented asynchronously using unique local identifiers. Ergo, the control flow for a message is restored when a response has been received. This is done by using the `twisted` threading library, conforming to the rest of the Tribler project.

In total the entire control message flow can be captured in 6 calls (reported here as they are defined in the code). These are the following, of which the first half are RPC calls from the main process to a child process and the latter half are RPC calls from a child process to the main process:

- `RPC_CREATE`: Set-up the child's TunnelCommunity
- `RPC_MONITOR`: Hidden services hash monitoring
- `RPC_CIRCUIT`: Try creating a circuit
- `RPC_SYNC`: Synchronize a shared object
- `RPC_CIRDEAD`: Forward that a circuit has died
- `RPC_NOTIFY`: Forward a generic notification from the child

These calls mostly speak for themselves, except for the synchronization of shared objects. What is done here is the forwarding of a change frame to the main process for visualization in the Graphical User Interface. This change frame is constructed by taking the shared objects (circuits and hops in the circuits) and monitoring the variable access to these objects. Once a variable has been changed/is dirty, it will be sent in a change frame. This change frame is sent periodically (by default,

every 5 seconds). This keeps the amount of calls and the amount of shared data to a minimum.

High-level Views

To separate concerns there is a functionality level built on top of the low-level views to provide functionality specific to management of a remote TunnelCommunity. Here the main view processes generic notifications, forwards synchronization calls, processes circuit deaths and hands over circuit return data to the main process's TunnelCommunity.

The child process, in this high-level view, is responsible for delivering data to its locally managed *TunnelCommunity* and sending data responses back to the parent over the data stream. The child process handles creation of the local TunnelCommunity, discovering peers interested in a certain torrent hash and circuit creation. All of the generated events are passed on to the main process, for visualization and high-level management. For all intents and purposes though, the child process is completely autonomous.

5.2.7 TunnelCommunity proxy class

Even though the design could function as it stands, another level of indirection has been added in the implementation. This is done to make the multi-core and the single-core architectures completely interchangeable according to the underlying hardware in the system. A proxy class for the TunnelCommunity has been implemented for the main process, providing the exact same functionality. This decorator is called the *PooledTunnelCommunity*.

This may seem useless, but becomes rather important when considering trivial variable access. Classes may contain static functions or attributes which can be *predicted* within the local process. For instance when dealing with circuits one may use a circuit id which was last updated several seconds ago. If the circuit no longer exists, which will happen rarely, data can be forwarded to the next available circuit as needed. This circumvents having to use aggressive circuit management, which could even also suffer the same problem.

Chapter 6

Experiments and Results

This chapter will discuss the experiments with the implementation as discussed in chapter 5. This chapter will also present the results of these experiments. These experiments will usually use Tribler's Gumby experiment framework to gather its results. This framework is made for large scale experiments and is therefore very suitable for a performance analysis of the given implementation.

The rest of this chapter will be structured as follows. First section 6.1 will lay out the experiments done with the message serialization, as described in section 5.1. Lastly, in section 6.2 and section 6.3 the experiments and results for the multi-core architecture implementation, as described in section 5.2, will be presented.

6.1 Serialization

Testing of the serialization implementation came in two flavors. First a lot of unit tests were added to the existing suites, providing just under a 25% increase in class coverage (line and statement coverage). Second, a large scale experiment was conducted, with profiling attached, to judge overall performance.

The first results from the unit tests showed a slight, but no significant performance benefit in the running time of the unit tests. The rest of this section will be devoted to the second experiment, also known as *the AllChannel experiment*.

6.1.1 The AllChannel Experiment

The AllChannel experiment involves running the Triber AllChannel community on 1000 Tribler instances, using 20 nodes. This is a default experiment shipped with the Gumby package. The nodes in this experiment are managed by the Distributed ASCI Supercomputer 5 (DAS5) server cluster on a single physical venue.

The AllChannel community itself is responsible for sharing *channels* between peers. Each of these channels form their own community, a *ChannelCommunity*. In turn, each of these channels share torrent files, playlists, moderations, comments and modifications. To provide load in the experiment, a single node publishes

torrent files in the experiment. At the start of the experiment this node publishes 497 torrent messages and will keep publishing 1 through 4 torrent messages for the following 150 seconds. The scenario is actually written for 3000 seconds, but this is to limit the system load. On another side note, the actual running time of the experiment is 350 seconds, but only 150 seconds remain for the actual scenario to run after set-up.

This test has been rewritten to allow for switching between the communities using serialization (BaseCommunity architecture) and all of the communities not using serialization. This allows for performance to be compared between the two implementations. The Gumby framework automatically collects data like RAM and CPU usage and Dispersy message synchronization statistics.

6.1.2 Results

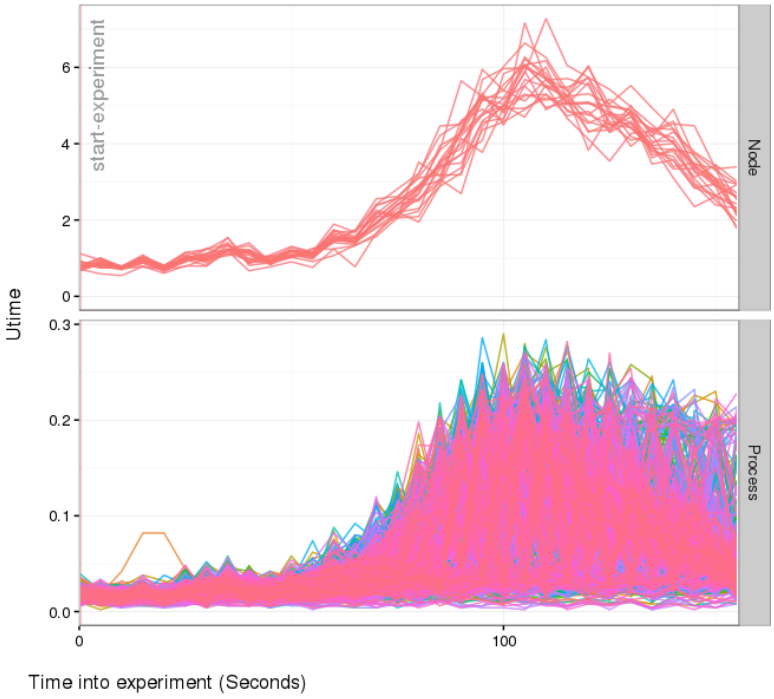
The first notable result of the new communities is that they require more RAM to be run by the application. However, this is a logical consequence of loading Google's `protobuf` library. A more interesting difference was in CPU consumption between the two implementations.

To explain these results the difference between the two CPU measures provided by Gumby must be explained. These are *utime* and *stime*. The *utime* measures the amount of time spent by a process being run in user space. One can think of the *utime* as the regular execution time of the process, being spent executing calculations on registers and RAM. In contrast, the *stime* measures the time spent by a process being run in kernel space. This means that this is the accumulation of time spent by the process handling Operating System level operations like disk I/O or interrupts. Both the *utime* and the *stime* are reported in hundreds of jiffies, which, for ease of explanation, we will refer to as simply the time units. A jiffy is actually one clock cycle of the CPU (so the clock frequency in Hertz of a machine is the amount of jiffies in a second).

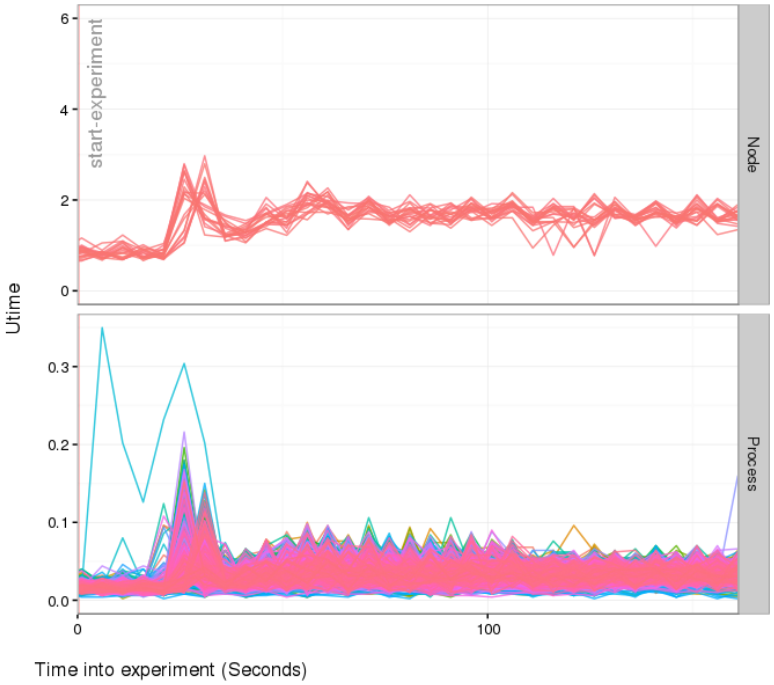
Both of the results of this experiment, which are presented in this section, have two graphs. One is the node (machine) aggregated CPU consumption, this is the top graph. This is the average of 50 processes running on the node. The bottom graph displays the results per process: this is useful for discovering excessive load on a single process.

Utime

First we will discuss the CPU time spent on user code, as presented in Figure 6.1. Chronologically, the first thing to take note of, for the user code, is the CPU time utilized by the initiating process. It can be observed that the CPU time spike for the initiating process in Figure 6.1b is much higher than the CPU time spike for the same process in Figure 6.1a (roughly 0.3 versus 0.1 units, in the per-process bottom-half graph). What this is telling of, is a much higher cost in CPU time for serialization of messages using the Protocol Buffers in comparison to the `struct`



(a) Old (struct) serialization



(b) New Protocol Buffers serialization

Figure 6.1: User code CPU times (utime)

packing. This is also logical, as the `struct` library does not perform any byte packing, whereas the Protocol Buffers library does.

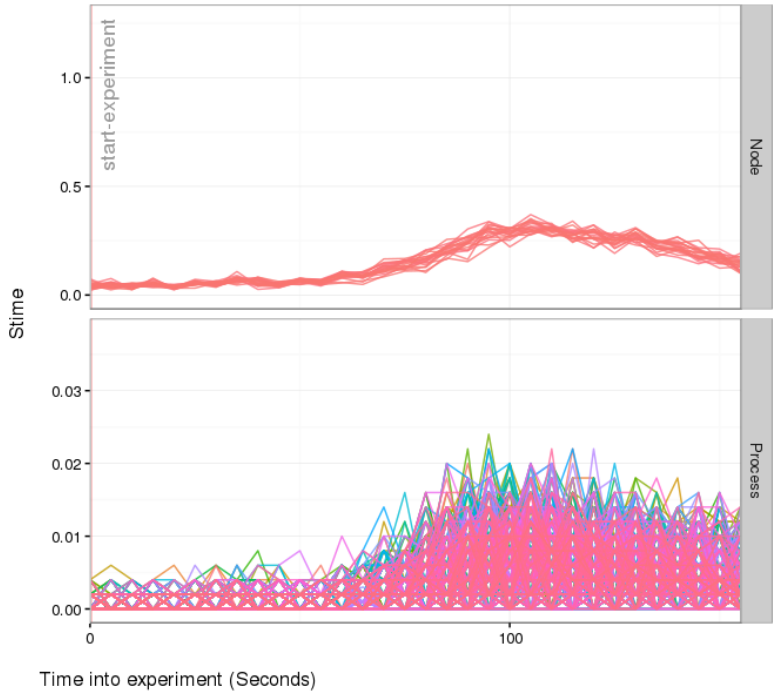
The second thing one notices, when looking at the cumulative CPU time usage *per node* (which are the upper-half graphs), is the enormous difference in CPU usage behavior. This phenomenon can be explained by two factors: the packing versus unpacking speeds of Protocol Buffers and the CPU scaling behavior as laid out in subsection 3.1.2.

The Protocol Buffers packing/serialization speed is much lower than the `struct` packing speed. This causes larger CPU usage spikes when packing messages, as shown by the higher CPU usage of the initiating process. As the initial batch of messages is shared with the first neighboring processes, Figure 6.1b also shows a bigger spike in the overall node CPU usage. This reactive spike to the first batch of messages is hardly visible in the old `struct` packing of Figure 6.1a.

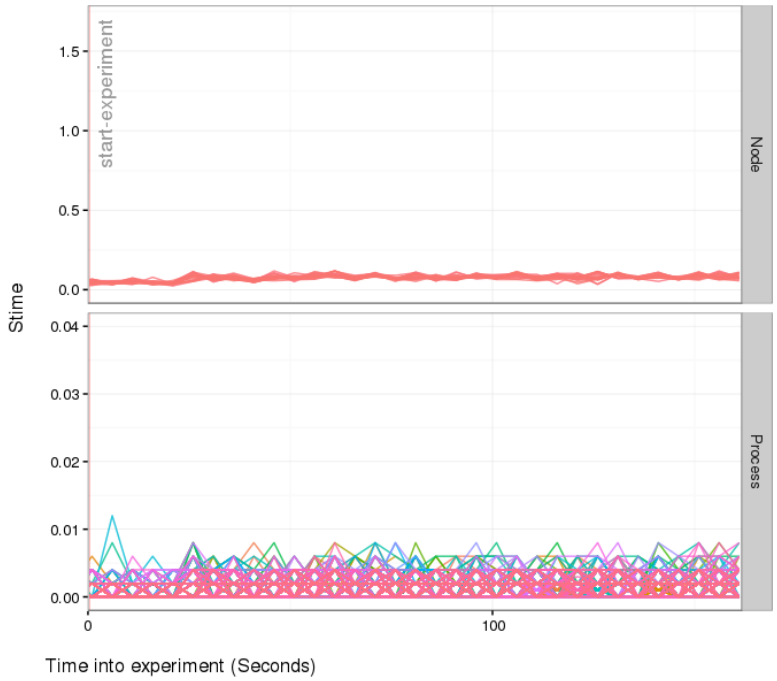
After this initial reactionary spike, the byte stuffing phenomenon from subsection 3.1.2 really starts influencing CPU usage. To reiterate, this phenomenon entailed that as the input buffers are saturated, the packing and unpacking time matter less than the message size reduction. This has the dramatic effect of a CPU usage reduction of about 80% under high network load. This is in the range of the aforementioned research, which found 59% and 90% decreases in CPU time (see chapter 3).

Stime

The next thing to look at is the CPU usage from system code, as presented in Figure 6.2. One can observe that the Protocol Buffers implementation (see Figure 6.2b) has a slight increase in usage at the first reactionary message serialization. This usage stabilizes at around 0.1 time units. This reactionary spike is (once again) all but invisible in the `struct` packing approach (see Figure 6.2a). The `struct` packing does peak higher when the network comes under load though. This is likely due to the *utime* CPU usage, causing the Operating System to interrupt more to free up CPU for process switching.



(a) Old (struct) serialization



(b) New Protocol Buffers serialization

Figure 6.2: System code CPU times (stime)

6.2 Architectural split

For the architectural split testing, a Gumby experiment was created. This experiment was performed on a 48-core 64 GB RAM server with a 1 GBps network adapter. Only local nodes were used (exit nodes, relay nodes, downloaders and seeders), isolating it from interaction with the Internet.

A 100 MB file was used per file transfer in the experiment, which provided several dozens of seconds of download time per transfer. This is almost the exact same setup as has been used by Ruigrok[17], with the exception of the file being transferred. In the experiment of this thesis the file was seeded with random bytes, in Ruigrok's case the file was filled with 0-byte values. The next subsection will discuss the exact experiment setup and will be followed by the results of the experiment.

6.2.1 Experiment setup

The experiment consisted of the following nodes:

- 1 Node with the new implementation running 16 workers/processes
- 1 Node with the new implementation running 32 workers/processes
- 1 Node with the new implementation running 48 workers/processes
- 4 Nodes seeding the random file
- 9 Nodes available for exiting the data in a circuit (candidate exit nodes), using the old single-core architecture

In the experiment the 4 seeders and 9 candidate exit nodes were statically present. The 3 downloading nodes with different amounts of workers were only loaded and utilized when a file transfer was initiated. After this they were unloaded as to not interfere with each other. This allowed for multiple file transfers within the same run of the experiment. Each file transfer was allocated 300 seconds to complete.

Early results showed that an abundance of seeders and exit nodes was beneficial for the download speeds. However, this also proved too much to handle for the experiment server. In fact, the multi-core implementation turned out to utilize so many resources the machine required a physical reset.

6.2.2 Download speed

In Figure 6.3 the download speeds (in kbps) are presented for the nodes running 16, 32 and 48 processes during a typical experiment. One can observe two behaviors in these results:

1. Download speed increases as the number of processes increases

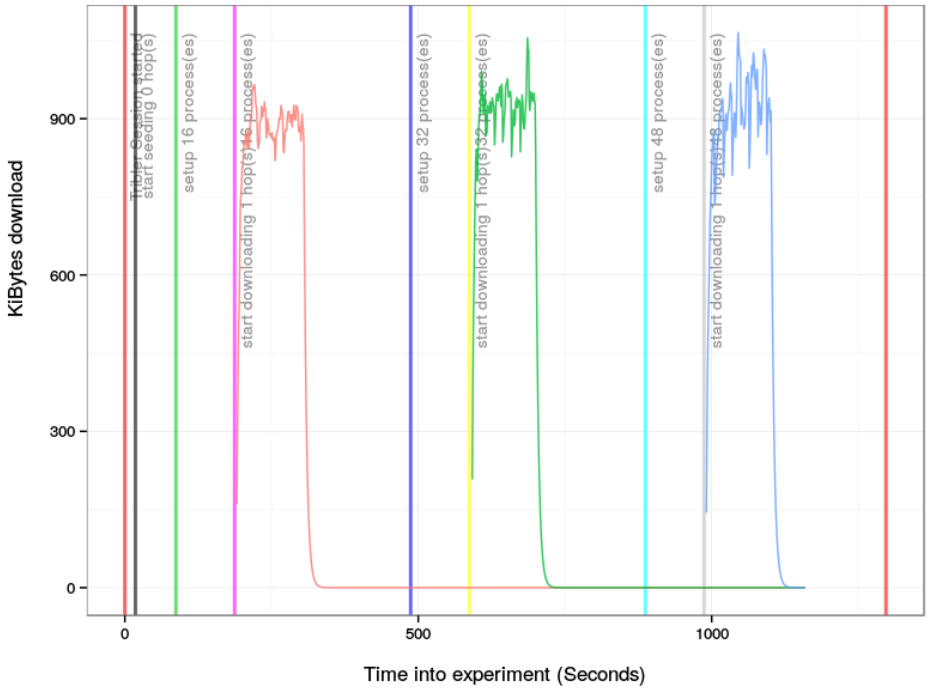


Figure 6.3: Download speeds for differing amounts of processes

2. Stability of the download decreases as the number of processes increases

These behaviors are noteworthy because behavior 2 is not in line with research conducted for multipath TCP, which suggests download speeds should become more stable and (at least) not slower when more paths are introduced[70].

Additional testing in both clinical environments and using the wild internet have however shown that this behavior stems from a shortage of seeding nodes. The unstable nature of the download speed seems to stem from the attention of too few seeders being divided over multiple downloaders (which are actually the multiple exit nodes used by the same downloader). In section 6.3 an experiment will be presented using the wild internet, showing this is indeed the case.

As a final note one might observe the slow download speed (roughly 0.9 Mbps) in this experiment. This is mostly due to the fact that the exit nodes are running the old architecture and the lack of seeders. Using a multitude of seeders, download speeds have been observed of 2.4 Mbps in the wild internet. Download speeds have been observed up to 18 Mbps when exit nodes also run the new multi-core architecture.

6.3 Architectural split in the wild

The wild internet was also utilized in the testing of the architectural split, to alleviate the bounds of isolated experiment execution as explained in the previous subsection. Experiments were performed on a 3.50 GHz quad core, 32 GB RAM physical machine with a 1 Gbps network adapter and Cat 6 Ethernet cabling. A fairly regular (slightly higher end) consumer machine. To control the available physical cores for testing, a virtual machine environment was used. The loaded Operating System on the virtual machine was Ubuntu 15.10.

Measurements are all based on 8 separate 2 minute runs using the same parameters. This was done to minimize the risk of 'getting lucky' when finding peers sharing a file. Some peers have higher bandwidth allocation than others, causing some peer selections to be more beneficial than others when testing. Overall these measurements spanned just over 20 hours (due to the author requiring sleep this was split up into one 14 hour and one 6 hour session). The experiment setup will be discussed in more detail in the following subsection, followed by the results of the experiments.

6.3.1 Experiment setup

For each of the measurements the procedure was as follows:

1. Remove existing Tribler settings and downloads
2. Start Tribler and initiate the designated download
3. Once the download has initiated its circuits and starts downloading, start measuring
4. After 2 minutes of downloading, stop measuring

Removal of Tribler settings was not strictly mandatory, but this was a convenient way to make sure the newly set defaults concerning the amount of processes used, were used by Tribler. Removal of downloads was done to make sure that chunks of the file with higher availability in the torrent swarm would persist for each experiment. In other words, if a highly available piece of a file has been downloaded by one experiment, the other experiment will be at a disadvantage, as other pieces of the file are not so easy to get.

To make sure that all experiments had equal opportunity to download, a single well-seeded file was selected. This selection consisted of using the most popular torrent tracker and selecting the most well seeded file. All experiments used this same file to download. The file remained seeded relatively the same throughout the experiment. The amount of seeders remained between 7150 and 7724 the entire time (mean: 7528.4, median: 7551). The amount of seeders was logged every 30 minutes, based on tracker info. Accuracy of these returned values may be questioned as several of the same values have been reported by the tracker (intuitively

the chance of 7544 seeders occurring 3 times in this process is rather low). At any rate, the actual number is not that important, as only a selection of 40 to 50 peers will be connected to each experiment.

Since it is already known that spawning more processes incurs more overhead, the measurements were only started when Tribler had created its required circuits. Application performance during circuit setup will remain outside of the scope of this thesis. This makes sure only the downloading portion of the application is tested. The downloading portion is the high throughput context this thesis aims to address.

Lastly, to collect the data presented in this thesis, every 1 second the aggregated CPU utilization percentage (over all cores) and the current download speed of the file were collected. For a 2 minute run this means that 120 data points were collected. This was done for virtual machines with full access to 1, 2, 3 and 4 3.5 GHz cores. The tested implementations were:

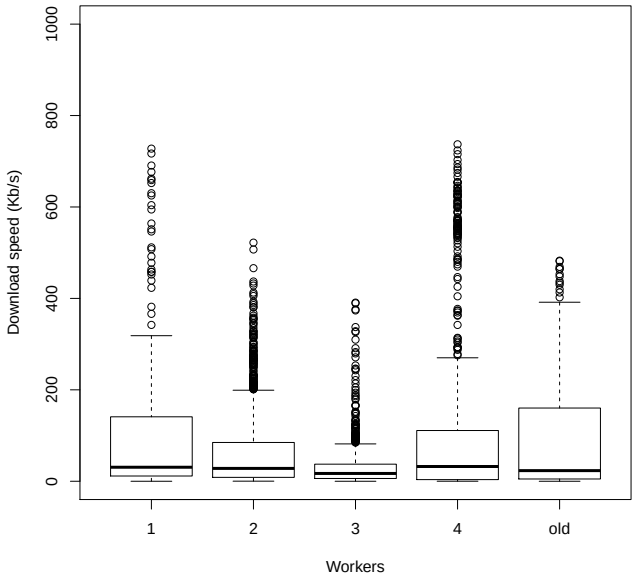
- 1 Child process multi-core architecture
- 2 Child process multi-core architecture
- 3 Child process multi-core architecture
- 4 Child process multi-core architecture
- The old single-core architecture

Each of these children (and the single-core architecture) was running the Tribler default of 4 circuits. So the applications for 1 through 4 processes had 4, 8, 12 and 16 circuits available respectively. Applications were also allowed to grow beyond this at runtime, up to double of the amount of circuits required before starting the download. This is also default behavior in Tribler.

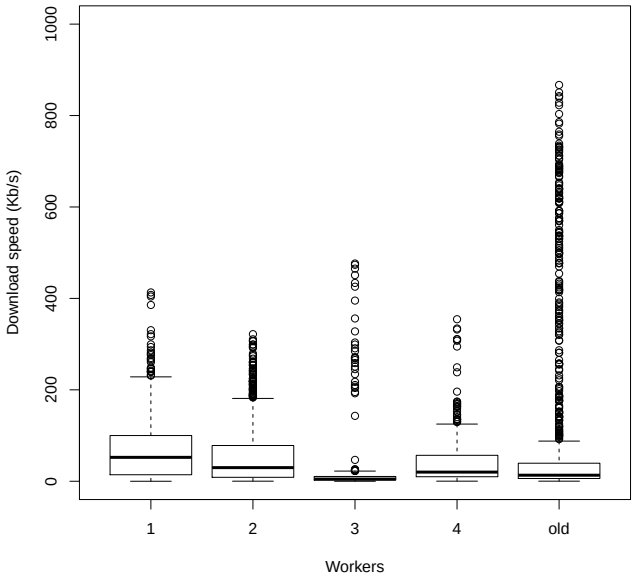
Finally, results have been filtered to strip measurements with either 0% CPU utilization or 0 KBps download speed. Overall this resulted in 18719 out of 21600 theoretical data points remaining. This means that 87% of the collected measurements were useful to this experiment. For easy viewing, this data will be presented throughout the rest of this section in box-and-whisker plots The rest of this section will also refer to the "X child process multi-core architecture" simply as an x *workers process* and the "old single-core architecture" as *the old worker*.

6.3.2 Download speed

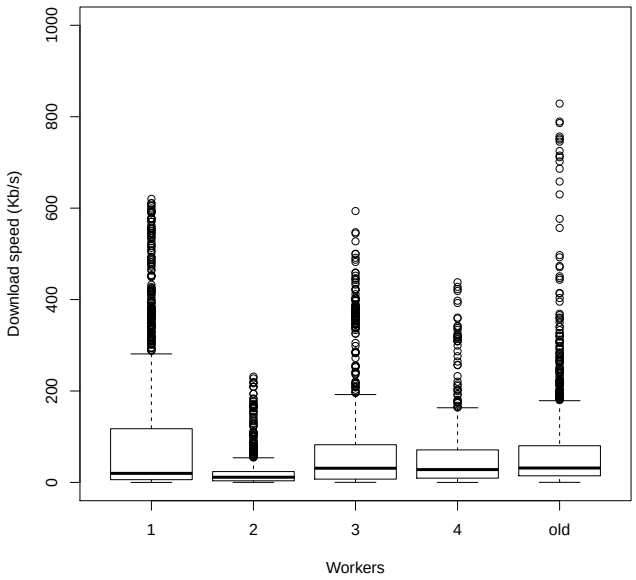
The acquired results in terms of download speed are been presented in Figure 6.4. Overall there are no significant differences in download speed, except for the 3 worker process on 2 cores (see Figure 6.4b) and the 2 worker process on 3 cores (see Figure 6.4c). What seems to have happened is that, even though measures were taken to prevent single experiments using high-speed peers, the overall speed



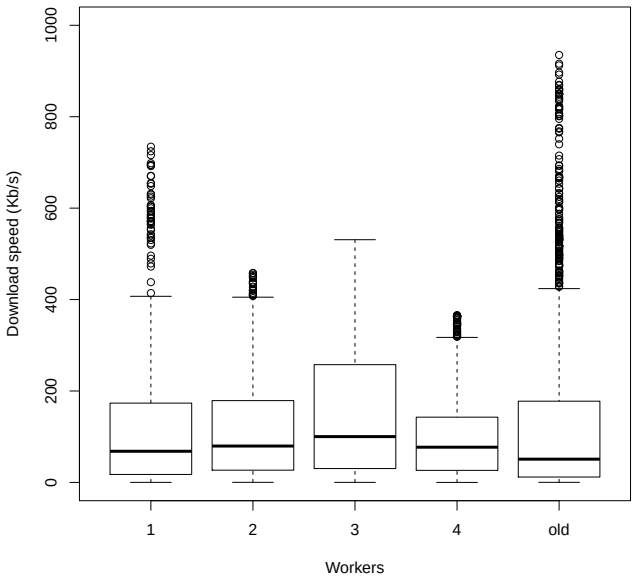
(a) 1 Physical Core



(b) 2 Physical Cores



(c) 3 Physical Cores



(d) 4 Physical Cores

Figure 6.4: Download speeds versus amount of workers

of the entire connected peer swarm was lower. This was not taken into account when designing the experiment.

Even though the mean value of the downloads speeds for a higher worker and higher physical core count seems to be prevalent, the fact that they are not universally significantly faster was a surprise. For instance, the 3 worker 4 core experiment only has a p-value of 0.2892 for having a higher download speed than the *old* worker 4 core experiment. That said, the is one significant speedup: the 4 worker 4 core experiment does achieve a statistically significantly higher download speed than the *old* worker 4 core experiment (with a p-value of 1.445×10^{-14}). Note that the reported p-values were calculated using Welch's t-test, thus the p-value reports the chance of equality of the datasets' means.

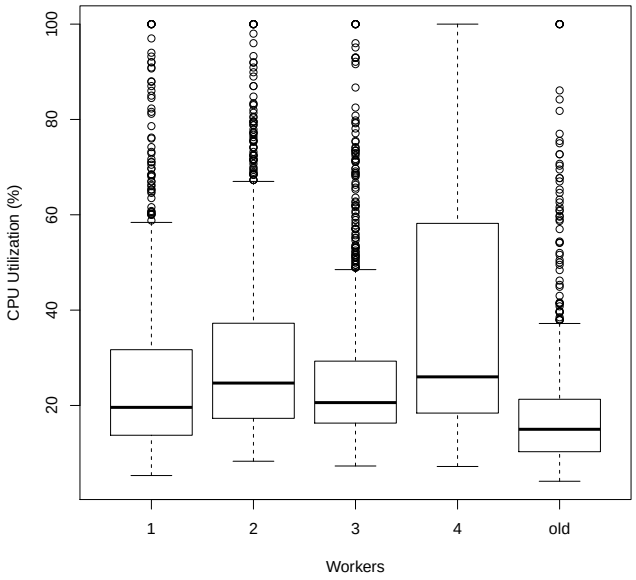
A final observation from Figure 6.4 is that, as the amount of cores increases, the (absolute) amount of data points which are higher in download speed than the box's whisker decrease. In other words, the download speed becomes more predictable, or stable if you will. This is in line with multipath communications research for TCP, suggesting a more stable connection when more paths are used[70]. The paths in this experiment are equivalent to the circuits used.

6.3.3 CPU consumption

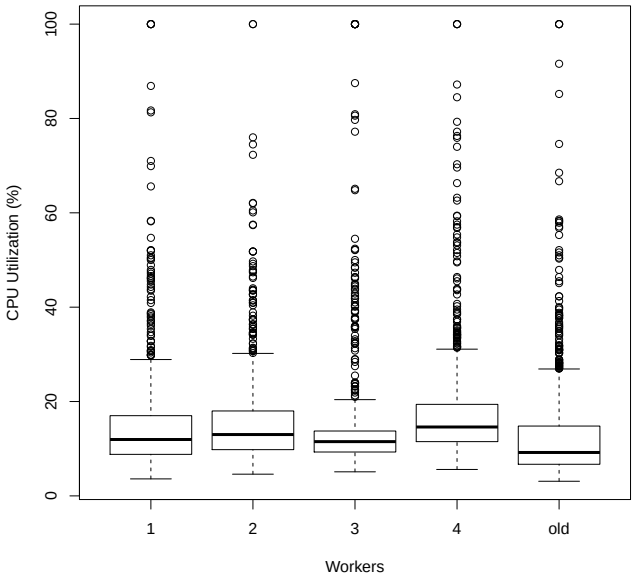
CPU utilization has been plotted in Figure 6.5. Note that the values in these graphs are in respect to the total CPU "power" available. Thusly it makes sense that as the amount of cores increases, the overall CPU usage lowers for the same experiments, as can be observed from the graphs.

From Figure 6.5a we can see that the core utilization is scaling as expected. The higher the amount of processes utilized, the higher the CPU utilization is. This leads up to the 4 worker 1 core scenario, where the whisker of the box plot is even touching the 100% mark. We also observe the expected behavior of the *old* worker, for which the CPU utilization drop proportionally as the amount of core increases. This is due to the fact that the *old* worker only uses a single core.

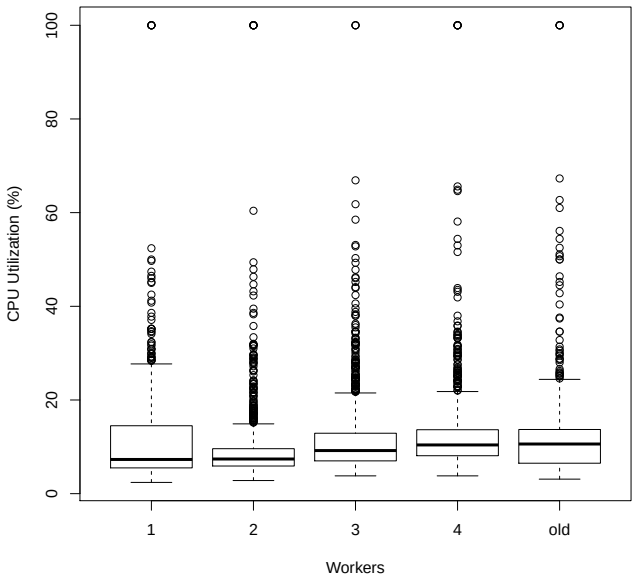
Interpreting Figure 6.5d we can determine the (median based) overhead costs of the 1 through 4 worker scenarios compared to the *old* worker scenario. These are 22.0%, 42.4%, 79.7% and 62.7%. This is interesting, as the amount of circuits is respectively 4, 8, 12 and 16 compared to the 4 of the *old* worker. In terms of scalability this would be a great result. This is also however, a bit unfair, as the amount of data passing through each of these applications differs (recall Figure 6.4). Therefore, in the next subsection we will look at a more fair metric.



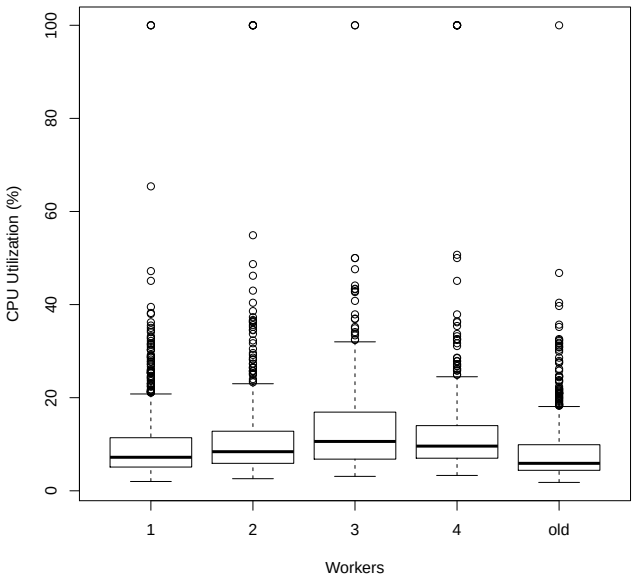
(a) 1 Physical Core



(b) 2 Physical Cores



(c) 3 Physical Cores



(d) 4 Physical Cores

Figure 6.5: CPU Utilization versus amount of workers

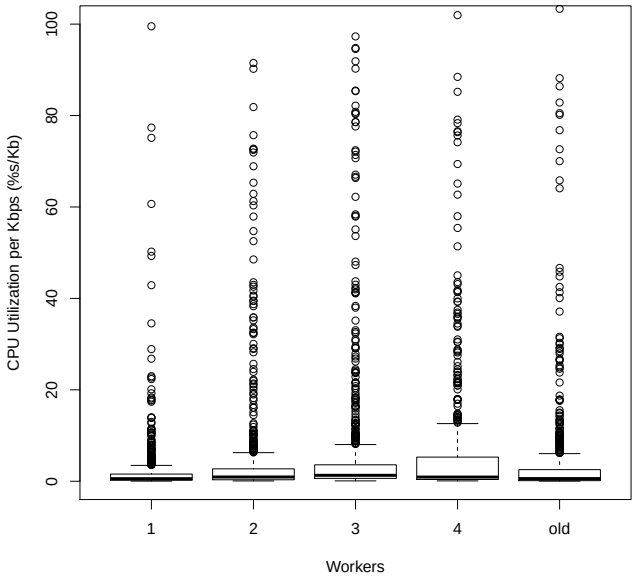
6.3.4 Efficiency

In an effort to alleviate the differences in download speed, we will also present the efficiency of the implementation. The efficiency can be described as the effort (CPU consumption) divided by the workload (download speed). Judging by the scaling behavior of Figure 6.5 we can assume the relation to be more or less linear. The results of this metric are presented in Figure 6.6: note the difference in y scales between the sub-figures. An important note here is that the CPU utilization per KB/s is given as it occurred at a certain time in the experiment, it is not a post-experiment calculation using the previously presented data. In other words, these are the unique $(CPU\ utilization, download\ speed)$ pairs as they occurred for the 5 different scenarios.

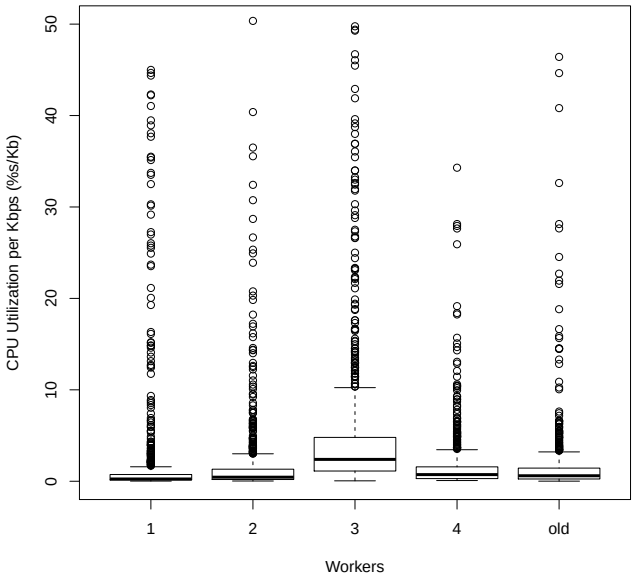
As previously noted, the download speeds for the 3 worker 2 cores and the 2 worker 3 cores scenarios were uncharacteristically low. We see the same in Figure 6.6b and Figure 6.6c, where the variance and median of both are very high.

Since the 4 core scenarios do not suffer from CPU choking or lowered swarm bandwidth, we will once again perform our median analysis on these. We now see a -5.9%, -7.2%, -17.6% and 10.9% increase in median utilization of the 1, 2, 3 and 4 worker scenarios compared to the *old* worker. So in comparison, the multi-core architecture (75% of the time) actually uses less CPU per kilobyte delivered per second.

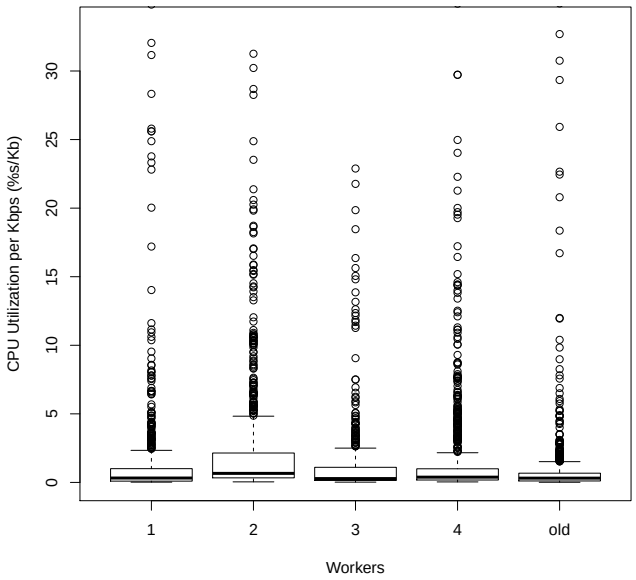
It is unclear what causes this dip in performance of the 4 worker experiment compared to the 1, 2 and 3 worker experiments. One of two things may be happening here. The first option is that the measurements were simply *unlucky*. In this case further research should be done with larger data sets and/or more workers and processors. The second option is that this is a turning point in performance. In other words, this could be the point where the overhead of extra processes becomes more than the CPU gain from parallelization. In this case further research should use more processors in experiments. Note that this turning point is expected at some point due to Amdahl's law.



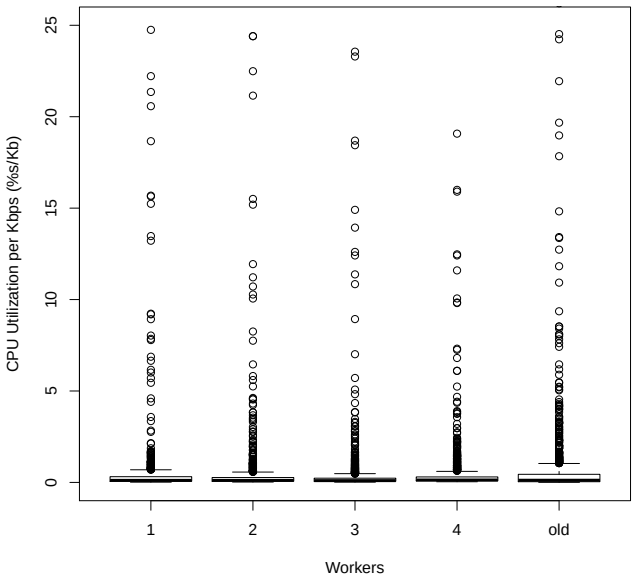
(a) 1 Physical Core



(b) 2 Physical Cores



(c) 3 Physical Cores



(d) 4 Physical Cores

Figure 6.6: CPU Utilization per Kbps versus amount of workers

Chapter 7

Conclusion

This thesis has modeled, predicted and confirmed key architectural opportunities for transitions to multi-core architectures for both distributed systems in general and anonymous file streaming applications in particular. In particular this thesis has contributed to the architecture design of these applications by:

1. Providing a solution for message handling scalability in high throughput contexts

2. Providing a method for scalable and performant architecture parallelization for high throughput contexts

This provides a roadmap for applications as to *how*, *why* and *what* parts of the architecture to change for optimal performance, maintainability and scalability.

Concretely, the two key components were: using message serialization with byte stuffing (or other compression) and cutting the single-core architecture for minimal data flow. The following sections will provide a quick summary of these findings.

7.1 Message serialization

Other research has shown that message serialization in distributed systems was wildly effective, with application CPU load being decreased by 59% through 90%. What has thus far been outside of the scope of scientific inquiry is why this is so wildly effective. This thesis has presented the compression (or packing) by byte-stuffing in input saturated systems to be the cause of this speedup.

In a normal sterile environment it has been shown that the speedup condition for message serialization is hard to satisfy. Scanning over a message and sending it, is also intuitively faster than scanning over a message, compressing it, sending it and decompressing it: judging sheerly by the number of steps involved. However, when the input buffer of the receiving node is emptied in parallel with the decompression of received messages the speedup condition changes. Now it is more easily satisfied and if the input buffer is guaranteed to be full for a significant

amount of time the speed of the compression and the decompression algorithm is nullified.

The implementation of serialization in Tribler had to be packed into existing structures for compatibility. It was decided that Google's Protocol Buffers was the best choice in terms of software compatibility, compression and format readability. Furthermore, issues with backward compatibility were presented and a simple but effective protocol to solve this issue.

When implemented in the Tribler platform, message serialization using Google's Protocol Buffers library also showed big changes. One noteworthy detail of this was that the packing time of the messages had a CPU load which was nearly tripled. In the end, under high load, this turned around completely, as expected. Under maximum load, the CPU load was, instead, one fifth of the original non-compressing implementation. In other words, this was a speedup of around 80%, which is in line with previous research.

The other noteworthy result is the approximate 2000 lines of code which were lost when switching to Protocol Buffers. It was found that Tribler used several custom serialization constructs which were intrinsically solved by the Protocol Buffers Domain Specific Language use. Most importantly, the cause of this was forced duplicate code.

7.2 Min-cut multi-core architecture extraction

A method to create a multi-core architecture from a single-core architecture for high throughput applications has been presented. This method involves profiling of the application to identify an application's component, which causes a high CPU load. Using a minimum s,t-cut in terms of data flow, one can create an optimal architecture with minimum overhead.

Furthermore, 3 data stream types have been identified, which will pass through these high throughput applications. These consist of *control*, *data* and *exit* flows. The data flows have been classed in terms of their Variety, Volume and Velocity (which are terms which were borrowed from the Big Data field and are very applicable). Due to the different characteristics of each of these data flows, it has also been determined that they should not be mixed.

The implementation of this multi-core architecture in Tribler brought several issues to light. These can mostly be attributed to software maintainability. For instance, the application should have a transparent architecture, instead of the non-transparent architecture resulting from a direct cut in the architecture. This transparency is not limited to improving maintainability, but also increases the scalability. The scalability stems from the scaling control based on real-time measurements which are possible in an Operating System (like current CPU load).

Another big problem in the implementation phase was the process creation. It was determined that Tribler was too complex for the memory re-use method called forking. Instead, spawning new processes was determined to be the only feasible

way to guarantee correct execution of the code.

Clinical trials were conducted using the multi-core implementation, showing an increase in download speed as the amount of deployed processes increased. Due to a relative shortage of seeding peers it was also shown, however, that the download speed was actually more unstable than single path communication. Consequently, when the amount of seeders was increased, the stability of a download was also shown to increase as the amount of processes/paths increased. The latter behavior was in line with existing research.

The results of the experiments show that there is a significant gain to be had from using a multi-core architecture over a single-core architecture. The amount of circuits which can be created with the same CPU utilization cost is, at least, more than linear. What has been identified as a possible point of diminishing returns for consumer hardware is 4 separate processes.

Bibliography

[1] Steve Crocker, D McMaster, and K McCloghrie. Host software. 1969.

[2] Thomas T Kwan, Robert E McGrath, and Daniel A Reed. Ncsa's world wide web server: Design and performance. *Computer*, 28(11):68–74, 1995.

[3] Phil Benson. *The Discourse of YouTube: Multimodal Text in a Global Context*. Routledge, 2016.

[4] Twitch. Twitch 2015 retrospective. 2016. Available at: `https://www.twitch.tv/year/2015`. Accessed November 3, 2016.

[5] Josh James. Data never sleeps 4.0. 2016. Available at: `https://www.domo.com/blog/data-never-sleeps-4-0/`. Accessed November 3, 2016.

[6] Max Rothschild. Corporate cyber-censorship: The problems with freedom of expression online. *Canadian Journal of Law and Technology*, 11(1), 2015.

[7] Philip Di Salvo. Strategies of circulation restriction in whistleblowing: the pentagon papers, wikileaks and snowden cases. *TECNOSCIENZA: Italian Journal of Science & Technology Studies*, 7(1):67–86, 2016.

[8] Annemarie Bridy. Internet payment blockades. *Fla. L. Rev.*, 67:1523, 2015.

[9] Mathew Ingram. Facebook's censorship of palestinian journalists raises serious questions. 2016. Available at: `http://fortune.com/2016/09/28/facebook-censorship-palestinian/`. Accessed November 3, 2016.

[10] Rima S Tanash, Zhouhan Chen, Tanmay Thakur, Dan S Wallach, and Devika Subramanian. Known unknowns: An analysis of twitter censorship in turkey. In *Proceedings of the 14th ACM Workshop on Privacy in the Electronic Society*, pages 11–20. ACM, 2015.

[11] Marjorie Heins. Brave new world of social media censorship, the. *Harv. L. Rev. F.*, 127:325, 2013.

[12] Marko Milosavljević and Sally Broughton Micova. Banning, blocking and boosting: Twitters solo-regulation of expression. *Medijske studije*, 7(13):43–57, 2016.

[13] Benjamin F Jackson. Censorship and freedom of expression in the age of facebook. *New Mexico Law Review*, 44(1), 2014.

[14] Kriti Arora. A study on privacy enhancing technologies for the internet. *International Journal of Advanced Research in Computer Engineering & Technology (IJARCET)*, 4(2):306–316, 2015.

[15] Mashael AlSabah and Ian Goldberg. Performance and security improvements for tor: A survey. Cryptology ePrint Archive, Report 2015/235, 2015. `http://eprint.iacr.org/`.

[16] Johan A Pouwelse, Pawel Garbacki, Jun Wang, Arno Bakker, Jie Yang, Alexandru Iosup, Dick HJ Epema, Marcel Reinders, Maarten R Van Steen, Henk J Sips, et al. Tribler: A social-based peer-to-peer system. *Concurrency and computation: Practice and experience*, 20(2):127, 2008.

[17] RJ Ruigrok. Bittorrent file sharing using tor-like hidden services. Master's thesis, TU Delft, Delft University of Technology, 2015.

[18] Quinten Stokkink, Harmjan Treep, and Johan Pouwelse. Performance analysis of a tor-like onion routing implementation. *arXiv preprint arXiv:1507.00245*, 2015.

[19] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. Technical report, DTIC Document, 2004.

[20] Vairam Arunachalam and William Sasso. Cognitive processes in program comprehension: An empirical analysis in the context of software reengineering. *Journal of Systems and Software*, 34(3):177–189, 1996.

[21] Harry M Sneed. Architecture and functions of a commercial software reengineering workbench. In *Software Maintenance and Reengineering, 1998. Proceedings of the Second Euromicro Conference on*, pages 2–10. IEEE, 1998.

[22] John J Marciniak. *Reengineering*. Wiley Online Library, 2002.

[23] Elliot J. Chikofsky and James H Cross. Reverse engineering and design recovery: A taxonomy. *IEEE software*, 7(1):13–17, 1990.

[24] Teodoro Cipresso and Mark Stamp. Software reverse engineering. In *Handbook of Information and Communication Security*, pages 659–696. Springer, 2010.

[25] Di Wu, Prithula Dhungel, Xiaojun Hei, Chao Zhang, and Keith W Ross. Understanding peer exchange in bittorrent systems. In *2010 IEEE Tenth International Conference on Peer-to-Peer Computing (P2P)*, pages 1–8. IEEE, 2010.

[26] Saikat Guha and Paul Francis. Characterization and measurement of tcp traversal through nats and firewalls. In *Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*, pages 18–18. USENIX Association, 2005.

[27] Andreas Müller, Georg Carle, and Andreas Klenk. Behavior and classification of nat devices and implications for nat traversal. *IEEE network*, 22(5):14–19, 2008.

[28] Niels Zeilemaker, Boudewijn Schoon, and Johan Pouwelse. Dispersy bundle synchronization. *TU Delft, Parallel and Distributed Systems*, 2013.

[29] LFD Versluis. Software performance engineering in complex distributed systems. Master's thesis, TU Delft, Delft University of Technology, 2016.

[30] Benjamin Frank, Ingmar Poese, Georgios Smaragdakis, Anja Feldmann, Bruce M Maggs, Steve Uhlig, Vinay Aggarwal, and Fabian Schneider. Collaboration opportunities for content delivery and network infrastructures. *Recent Advances in Networking*, 1:305–377, 2013.

[31] B Pourebrahimi, K Bertels, and S Vassiliadis. A survey of peer-to-peer networks. In *Proceedings of the 16th Annual Workshop on Circuits, Systems and Signal Processing, ProRisc*, volume 2005. Citeseer, 2005.

[32] Al-Mukaddim Khan Pathan and Rajkumar Buyya. A taxonomy and survey of content delivery networks. *Grid Computing and Distributed Systems Laboratory, University of Melbourne, Technical Report*, page 4, 2007.

[33] Krista Bennett and Christian Grothoff. Gap–practical anonymous networking. In *International Workshop on Privacy Enhancing Technologies*, pages 141–160. Springer, 2003.

[34] David L Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.

[35] Bassam Zantout and Ramzi Haraty. I2p data communication system. In *Proceedings of ICN*, pages 401–409, 2011.

[36] Oliver Berthold, Hannes Federrath, and Stefan Köpsell. Web mixes: A system for anonymous and unobservable internet access. In *Designing Privacy Enhancing Technologies*, pages 115–129. Springer, 2001.

[37] Jean-François Raymond. Traffic analysis: Protocols, attacks, design issues, and open problems. In *Designing Privacy Enhancing Technologies*, pages 10–29. Springer, 2001.

[38] Ye Zhu, Xinwen Fu, Bryan Graham, Riccardo Bettati, and Wei Zhao. Correlation-based traffic analysis attacks on anonymity networks. *IEEE Transactions on Parallel and Distributed Systems*, 21(7):954–967, 2010.

[39] Keith D Watson. Tor network: A global inquiry into the legal status of anonymity networks, the. *Wash. U. Global Stud. L. Rev.*, 11:715, 2012.

[40] Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *Distributed Shared Memory-Concepts and Systems*, pages 42–50, 1991.

[41] František Plášil and Michael Stal. An architectural view of distributed objects and components in corba, java rmi and com/dcom. *Software-Concepts & Tools*, 19(1):14–28, 1998.

[42] Jürgen Müller, Martin Lorenz, Felix Geller, Alexander Zeier, and Hasso Plattner. Assessment of communication protocols in the epc network-replacing textual soap and xml with binary google protocol buffers encoding. In *Industrial Engineering and Engineering Management (IE&EM), 2010 IEEE 17Th International Conference on*, pages 404–409. IEEE, 2010.

[43] Jianhua Feng and Jinhong Li. Google protocol buffers research and application in online game. In *Conference Anthology, IEEE*, pages 1–4. IEEE, 2013.

[44] Google. Protocol buffers. 2016. Available at: `https://developers.google.com/protocol-buffers/`. Accessed November 9, 2016.

[45] Sandstorm.io. Cap'n proto: Introduction. 2016. Available at: `https://capnproto.org/`. Accessed November 9, 2016.

[46] Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.

[47] Oracle. An overview of rmi applications. 2016. Available at: `https://docs.oracle.com/javase/tutorial/rmi/overview.html`. Accessed November 9, 2016.

[48] Steven Raemaekers, Arie Van Deursen, and Joost Visser. Semantic versioning versus breaking changes: A study of the maven repository. Technical report, Delft University of Technology, Software Engineering Research Group, 2014.

[49] Mouaaz Nahas, Michael Short, and Michael J Pont. The impact of bit stuffing on the real-time performance of a distributed control system. *CAN in Automation*, pages 101–107, 2005.

[50] Douglas C Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*, volume 2. John Wiley & Sons, 2013.

[51] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[52] Bryan Cantrill, Michael W Shapiro, Adam H Leventhal, et al. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference, General Track*, pages 15–28, 2004.

[53] Alex Zakonov and Victor Mushkatin. Exposing application performance counters for. net applications through code instrumentation, November 1 2011. US Patent 8,051,332.

[54] Alexander Roghult. Benchmarking python interpreters: Measuring performance of cpython, cython, jython and pypy. 2016.

[55] Deepa Viswanathan and Sheng Liang. Java virtual machine profiler interface. *IBM Systems Journal*, 39(1):82–95, 2000.

[56] Dale Skeen and Michael Stonebraker. A formal model of crash recovery in a distributed system. *IEEE Transactions on Software Engineering*, (3):219–228, 1983.

[57] Daniel Bilar and Daniel Burroughs. Introduction to state-of-the-art intrusion detection technologies. In *Enabling Technologies for Law Enforcement*, pages 123–133. International Society for Optics and Photonics, 2001.

[58] Jack Shih-Chieh Hsu, Chien-Lung Chan, Julie Yu-Chih Liu, and Houn-Gee Chen. The impacts of user review on software responsiveness: Moderating requirements uncertainty. *Information & Management*, 45(4):203–210, 2008.

[59] Van Jacobson, Robert Braden, and David Borman. Tcp extensions for high performance. Technical report, 1992.

[60] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[61] Christof Fetzer and Zhen Xiao. A flexible generator architecture for improving software dependability. In *Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th International Symposium on*, pages 102–113. IEEE, 2002.

[62] Wikipedia. List of tcp and udp port numbers. 2016. Available at: `https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers`. Accessed November 12, 2016.

[63] Paul Zikopoulos, Chris Eaton, et al. *Understanding big data: Analytics for enterprise class hadoop and streaming data*. McGraw-Hill Osborne Media, 2011.

[64] Johan Pouwelse, Paweł Garbacki, Dick Epema, and Henk Sips. The bittorrent p2p file-sharing system: Measurements and analysis. In *International Workshop on Peer-to-Peer Systems*, pages 205–216. Springer, 2005.

[65] Burkhard Monien and Ivan Hal Sudborough. Min cut is np-complete for edge weighted trees. *Theoretical Computer Science*, 58(1):209–229, 1988.

[66] Lester R Ford and Delbert R Fulkerson. Maximal flow through a network. *Canadian journal of Mathematics*, 8(3):399–404, 1956.

[67] Michael T Goodrich and Roberto Tamassia. Algorithm design. *Wiely India*, 2002.

[68] Marcus Leech, Matt Ganis, Y Lee, Ron Kuris, David Koblas, and L Jones. Rfc 1928: Socks protocol version 5. Technical report, RFC, IETF, March, 1996.

[69] Li Li and Allen D Malony. Model-based performance diagnosis of master-worker parallel computations. In *European Conference on Parallel Processing*, pages 35–46. Springer, 2006.

[70] Yung-Chih Chen, Yeon-sup Lim, Richard J Gibbens, Erich M Nahum, Ramin Khalili, and Don Towsley. A measurement-based study of multipath tcp performance over wireless networks. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 455–468. ACM, 2013.