INF-3990

Master's Thesis in Computer Science

# A Client for Chain Replication

Ingar Mæhlum Arntzen

December, 2009

Faculty of Science
Department of Computer Science
University of Tromsø

## Acknowledgements

The author wants to thank Fred B. Schneider and Robbert van Renesse (Cornell University) for leading and inspiring my work with the chain replication protocol. Furthermore I would like to thank Fred B. Scheider for inviting me as a visiting scientist to Cornell for one year, and Dag Johansen (University of Tromsø) for facilitating the visit and funding it through the [WAIF] project. Anders Andersen (University of Tromsø) has been my advisor during the write-up of this thesis. Finally, Norut - represented by Lars K. Vognild and Njål T. Borch - supplied me with an office, a desk, a computer, and very good coffee.

# Table of Contents

# 1. Introduction

Design of a distributed, reliable storage systems generally calls for certain tradeoffs to be made. Especially, properties such as low latency, high throughput, scalability, fault-tolerance, strong consistency and high availability may represent conflicting goals for the system designer. A typical engineering approach to this problem might be to first focus on the performance of the system. Then, after the performance objectives have been met, implement non-functional requirements - often in a best-effort manner.

In Chain Replication [CR] this thinking is reversed. The primary goal is to offer the unusual combination of strong consistency and high availability, and then to minimize the negative effects this might have on performance. This motivates the rather unorthodox architecture of chain replication. In chain replication, as the name suggests, storage nodes are organized linearly, forming a directed chain. Updates are replicated sequentially along the chain.

Sequential update propagation means that update latency is going to be high. However, despite this weakness, a linear topology has attractive properties too.

- Strong consistency can be implemented by ensuring that all Queries are processed by a single node in the chain, the last one. Sequential update propagation implies that clients only see the effects of successfully replicated updates.
- High availability depends on fault-tolerance and quick recovery. The strict linear topology is a good basis for protocols implementing both fault-tolerance and efficient recovery.
- High throughput is possible even though update latency is high. The linear topology allows updates to be efficiently pipelined through the chain.

So, chain replication offers an interesting tradeoff, trading update latency for the combination of strong consistency, high availability and high throughput [CR]. This of course is especially attractive in applications where short update latency is not a critical requirement. In [CR] search engines are emphasized as one application type that fits this description.

The original paper on chain replication [CR] gives an abstract presentation of the chain-replication protocol, along with an analysis of its central properties. So far though, a real system implementation has not been presented. Here we aim to make a contribution in this respect. However, a full implementation of a chain replicated storage-system is a huge undertaking. Therefore, in this thesis we will limit ourselves to addressing a particular part of chain replicated storage system; the *chain client*.

## 1.1 The Chain Client

The architecture of a chain-replicated storage system, as presented in [CR], is comprised of three parts; the *chain*, the *master* and the *chain client*.

- The chain is a linear arrangement of storage nodes. Updates are received by the first node of the chain, the Head, and then processed and propagated sequentially along the chain, until they reach the last node, the Tail. Queries are processed by the Tail only.
- The master ensures the integrity of the system in the event of failures. It detects failures in the chain and coordinates the recovery protocol. Additionally, failures in the chain may cause clients to become disconnected from the chain. For this reasone the master must also coordinate the re-connection of clients in such circumstances.
- The chain client interacts directly with the chain, by issuing requests (Updates and Queries) and receiving replies (Query-Replies and Update-Replies). The chain client also interacts with the master as part of the recovery protocol. The chain client may be thought of as the application client.

In this thesis we focus on the design and implementation of a chain client. The chain itself may be the central challenge. Still, as will be clear, the client is not trivial either.

First, the design and implementation of the chain client is not an isolated concern. To the contrary, the chain client is in some sense an integral part of the chain replication protocol. As indicated above, the chain client interacts directly with both Head and Tail in order to make use of the chain. In particular, after certain failures in the chain (Head/ Tail failure), the chain clients will have to collaborate both with the master and the chain nodes in order to reconnect to the (recovered) chain.

Second, the details of the interfaces between the chain, the master and the chain client are not a priori defined. Instead, it is the exercise of designing and implementing the chain client that will allow us to suggest appropriate interfaces. Note also that the design and implementation of the chain and the master are relevant too, with regard to the design of the interfaces. For this reason we will try as much as possible to include the concerns of the chain designer into our design discussions. Here we benefit from prior experience with an early prototype of the chain.

Third, not even the system architecture is written in stone. In the paper [CR] the master is defined to be a reliable process, *external* to chain. Note however that this is so primarily in order to simplify explanation. Especially, it would be possible to have the responsibilites of the master reliably distributed along the chain. This is an attractive idea since it would simplify the entire system architecture a great deal.

## 1.2 Goals

With a basic understanding of the chain client, the role it plays in the system, and the design space available with respect to system architecture and interfaces, we are ready to state some goals for this thesis.

> **1) Design and Implement a Generic Chain Proxy**. All chain clients are much involved with the chain protocol and therefore they have much logic in common. We aim to ease the implementation of chain clients (i.e. chain applications), by encapsulating and hiding shared complexity within a *chain proxy*. This way, the chain proxy may empower application programmers by giving them a higher-level API to the chain. We call this the *chain API*. As part of defining the chain proxy and the chain API, it will also be necessary to define the *client-chain interface*.
>
> **2) Remove the Client-Master Dependency**. As argued above, the responsibilities of the master may be migrated into the chain. We want to encourage this by removing the dependency between the chain client and the master. This means that the chain client will have no knowledge of the master, it will only interact directly with the chain. This has implications for the design and implementation of the chain.[1] In particular the chain must manage *client-sessions*, including the recovery of client-sessions after failures. A *session-protocol* is required to implement this reliably within the chain.

## 1.3 Problem Statement

The problem statement of this thesis is derived from the above goals.

1. Design a session-protocol that ensures the integrity of client-sessions even in the event of chain failures (i.e. Head-failure and Tail-failure). The protocol shall be executed by the chain clients and the chain exclusively (i.e. not require the master to participate).
2. Design and implement a chain proxy.
    a. The chain proxy shall encapsulate functionality common to all chain clients.
    b. The chain proxy shall provide programmers with a chain API in order to ease the development of chain-replicating applications. The chain API shall:
        1. provide flexibility and completeness.
        2. support a variety of applications types and programming models.
    c. The chain proxy shall implement the session-protocol defined in 1).
    d. The chain proxy shall strive to be as thin as possible. By this we mean:
        1. Minimise the overhead imposed by the proxy (latency and throughput).
        2. Expose the true behaviour of the chain, as opposed to masking it.

1. This will not necessarily require the chain to carry all the responsibilities of the master, only those responsibilities relevant to chain clients.

## 1.4 Scope

As mentioned above, the scope of this thesis does not include the design and implementation of the chain-replication protocol itself. This is because we consider the challenges of the the chain proxy to be sufficiently demanding for a Master's Thesis. Still, we will need a chain in order to test our implementation of the chain proxy. For this purpose we have implemented a *chain simulator*. Essentially this is a chain of length one, thus it plays the role of both Head and Tail simultaneously. The chain simulator implements the client-chain interface expected by the chain proxy, including the session-protocol.

Also, as part of this project we have created a custom network abstraction called Message Ports. This abstraction encapsulates much complexity related to non-blocking socket operations, io-scheduling, connection failures and message transfer. This network abstraction is not essential to the project, in the sense that the project could have been completed without it. Still, we have chosen to give the Message Port abstraction a prominent place in this thesis. There are several reasons for this.

First, our implementation of Message Ports is the basis for the implementation of both the chain proxy and the simulator. The chain proxy inherits many important properties from Message Ports, both functional and non-functional. Second, the Message Port abstraction is helpful as an explanatory tool. For instance, our presentation of the core logic of the chain proxy is much simplified by the introduction of Message Ports. Third, the Message Port abstraction solves problems that are equally relevant for nodes in the chain. Thus, an implementation of the chain replication protocol could benefit much from this abstraction too.

## 1.5 Overview

The rest of the thesis is organized as follows.

- Section 2 discusses reliable storage systems in general.
- Section 3 presents the chain-replication protocol.
- Section 4 presents the chain-client. We define the chain-proxy by discussing how it interacts with chain-applications and with the chain.
- Section 5 presents our design for the session-protocol.
- Section 6 presents Message Ports, our network programming abstraction.
- Section 7 discusses the design of the chain proxy.
- Section 8 presents the implementation of Message Ports.
- Section 9 presents the implementation of the chain proxy.
- Section 10 discusses application development and presents a real application, a chain-replicated filesystem.
- Section 11 presents the chain simulator and evaluates our implementation of the chain proxy.
- Section 12 summarises the thesis and gives concluding remarks.
- Appendix: Code Overview

# 2. Background

Reliable storage services are a crucial component in most computing systems, and the challenge of providing reliable, persistent storage is addressed on many levels. Hard drive manufacturers employ a host of tricks to improve the reliability of a single disk. Still, efforts to produce highly reliable hard drives must be balanced against the alternative, implementing storage reliability by means of multiple not-so-reliable, not-so-expensive hard disks.

This is the core idea behind RAID[2] [RAID] technology. Multiple disks are put together to implement the abstraction of one single large disk. Now, by reserving some of the accumulated disk space for redundant data, this large disk may be made more reliable than any of its internal, physical disks. This may be achieved by mirroring data on multiple physical disks. This protects from disk failures, but implies that 50% of the disk space is redundant (if the replication factor is 2). Parity is an alternative technique that in effect only requires 1 physical disc (out of N) to be reserved for redundant data. This means less redundancy, but also less reliability. If more than one disk fail simultaneously, all data will be lost.

RAID is typically used to provide reliable storage within a single machine. However, this idea of implementing reliable storage by replicating data across multiple not-so-reliable disks, is also the core idea of all distributed storage services. In a distributed storage service physical disks are distributed across multiple machines connected by some network, i.e. a LAN or a WAN. This way storage reliability may be improved even further. When data is stored reliably within a single machine, data can still be lost due to devastating events, such as fire or explotion. In contrast, if data is replicated across the world, very strong guarantees can be made with regard to data safety.

Distributed storage systems come in many shapes; Distributed reliable Filesystems[HDP, GFS] and Databases. Distributed storages systems based on P2P technology, i.e. DHT's [PAST]. In additon there are some well known commercical solutions, such as GoogleFs [GFS], Microsoft and Amazon. Most of these systems can be characterised as middleware, in the sense that replication is implemented in software (as opposed to hardware) and that the complexities introduced by replication are mostly hidden from the application programmers.

As a background for this thesis, we will discuss general properties of such distributed, reliable storage services. In the next section, we will consider a specific instance of such a system, Chain-Replication.

## 2.1 Distributed, Reliable Storage Services

Storage services store data objects reliably. Data objects may be files, records, data-chunks and more, depending on the type of system. The reliability usually comes from replication of such data objects across multiple machines. A storage service may be comprised of N storage nodes, where each data object is replicated across a subset M of these nodes. M is in this case the replication factor. Ideally, there are always M replicas of each data object, and no two replicas are stored at the same physical machine.

Now, these data objects are replicated for a reason. They are valuable to the clients of the storage service. A distributed storage service may support a large number of clients accessing the data objects, possibly simultaneously. All storage systems distinguish two

2. Redundant Arrays of Inexpensive (or Independent) Disks

types of client access.

> **Read**. A client may read a data object. Typically this means that the service returns a copy of the data object. By returning a copy, rather than a reference to the real data object, the client is given access to the content of the data object, without being able to change it in any way. This type of operation may also be know by other names, e.g. query, get, fetch, lookup or request.
>
> **Write**. If the client wants to change the content of the data object in any way, this ability is usually provided by a write operation. It may also be know by a host of other names such as update, change, set, append, push, pop, truncate, replace, insert, create, add, remove, delete. Such operations alter the content of data objects. Thus, in order to keep all replicas identical, if a write operation is applied to one replica, it must be applied to all.

When designing a distributed, reliable storage service there are many issues that need to be addressed. Here are some prominent design points.

**Scale**. Scaleability may be desired along several axis, for example in number of storage nodes (N), in aggregate storage space or in number of clients. Hadoop[HDP] and GoogleFs[GFS] are reliable file systems that achieve impressive scalability in all these dimentions. This ability for scale is largely due to a very limited write access to data objects.

**Replication factor**. How many replicas (M) should be maintained per data object. This is mainly a tradeoff between available storage space and required level of reliability. Hadoop allows the client application to set this parameter on a per-object basis.

**Replica placement**. If N>M, a policy may direct how M replicas are distributed across N nodes. In certain DHT based replication schemes such as [PAST, CFS] replica placement is random. This may be because the location of a data-object is coupled the DHT adressing scheme, which by design aims for randomness. On the other hand, OceanStore[OCN] is able to improve the locality by keeping data-objects close to clients. Hadoop and GoogleFs are optimized for read access, implicating that properties of the underlying network topology must be taken into account. For this reason, both systems implement replica placement policies with *rack-awareness*.

**Semantics of Write Operations**. Operations can be primitive or sophisticated. A primitive write mechanism would require data objects to be replaced, in order to be changed. Hadoop implements a write-once read-many policy, thereby requiring data-objects to be deleted and recreated, in order to be changed. GoogleFs do implement the regular filesystem write operation, but it is not optimized for this purpose. Instead, it is optimized to allow many clients to append the same file, concurrently. Chain Replication[CR] supports a much more expressive update mechanism. An Update is an operation which is applied to a single data-object. The processing of an update may take as input multiple parameters along with the current state of the data-object. Chain Replication may even allow for non-deterministic calculations as part of Updates.

**Coordination of Read Operations**. There are multiple ways in which the service may resolve a read operation. In GoogleFs and Hadoop the client queries a sentralised service in order to obtain the location of data-objects. This also includes all replicas. The client may then choose to read directly from the closest replica. So, in general read operations may be performed on any replica. Again, Chain Replication is different because all read operation go to one particular replica. Other systems such as [PAST, CFS] use cashing to reduce read latency.

**Coordination of Write Operations**. Write operations need to be applied to all replicas. There are many strategies for achieving this. If a system is designed according to the state machine approach [SMA], the client will  issue write requests directly to all replicas, in parallel. The primary-backup approach [PBA] is a centralized architecture where one replica plays the role of primary, and the rest are backups. All client write requests then go through the primary, which forwards each request in parallel to all backups. Both these approaches typically apply write operations to replicas in parallell. However, write operations can also be applied in a serial/linear fashion. In Hadoop the client writes one replica directly. The rest of the replication is performed lazily and serially by the system. In GoogleFs the data associated with a write is propageted serially to all replicas. Still, the write operation is committed in parallell at all replicas. In Chain Replication updates are propagated serially, and the operation does not complete until all replicas are updated.

**Consistency**. If multiple read and write operations are simultaneously applied to the same data object, by multiple clients, there is a potential for both conflict and confusion. Failures too may be a source of inconsistencies.  A consistency model is a specification of service behaviour in such circumstances. In a weak consistency model few guarantees are made by the system. This makes the system easy to build, but it makes life more difficult for the programmer. In contrast, a system with a strong consistency model will try to mask some of the complexity that originates from concurrency of clients.

Google Fs implements a very relaxed consistency model. The append operation is guaranteed to be atomic. All clients see the same ordering of appends, on all replicas, but that order may not reflect any valid serialisation of the client operations; Appends may be duplicate or there may be padded areas in the result file. However, these issues can be resolved by the applications using the system.

On the other end of the specter, Primary-Backup protocols (blocking) may be used to support *strong consistency*. This means that a read (query) operation issued to the system will always reflect the effects of any write (update) operation that preceded it. This is possible by implementing a global ordering of all operations (reads and writes) from all clients. Chain Replication implements strong consistency by having all operations ordered by one single machine in the chain (the Tail).

**Failure-detection and Recovery**. When a storage node is lost, many replicas of many objects are lost with it. So, in order to maintain the ideal of M replicas per data object, new replicas must be created as a response to failure. In Hadoop, a hearbeat mechanism is used to detect node failures. In the event of a failure, the system immediately starts re-replicating data-objects (file blocks) while the rest of the replicas remain available. Still, in Hadoop the filesystem namespace manager is centralised, making it a single-point of failure.

Primary-backup protocols may provide better fault-tolerance by avoiding single-point of failure all together. This is possible by including a distributed election protocol to select a new primary, every time the primary fails. Unfortunately, distributed coordination protocols of this sort generally do not scale well, and may be time-consuming. Chain Replication too avoids single point of failure, but does not require any election protocols. Also the steps required to recover after a failure in the chain need not be time-consuming.

## 2.2 Design Objectives, Tradeoffs and Dependencies

As indicated above, these design points are not independent of each other. Rather, some of them directly conflict, so in general the designer of a storage service must make some compromises. To some degree this explains the variety of distributed, reliable storage systems.
Especially, scalability and performance seem to be at odds with properties such as high levels of consistency and fault-tolerance. Furthermore, high availability requires fault-tolerance, but may have been difficult to combine with strong consistency.

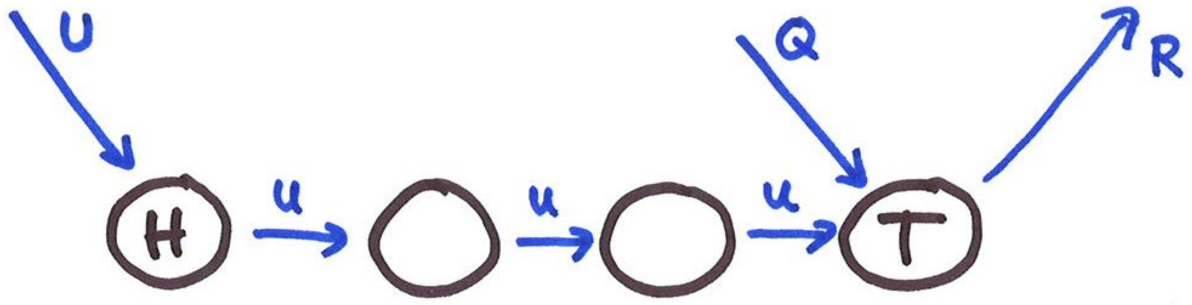
Chain replication is interesting because it represents an unusual tradeoff between all these properties. It promises to combine strong consistency with high availability, fault-tolerance and even high throughput. The downsides of this protocol may be that update latency is high and that query processing is not distributed along the chain.

# 3. Chain Replication

This section gives a detailed presentation of the Chain Replication protocol.

## 3.1 The Protocol

The Chain Replication Protocol is illustrated by the figure below. Replica nodes are set up in a linear fashion, forming a chain. The first node in the chain is called the Head (H). The last node is called the Tail (T). The Head of the chain accepts Update-requests (U) from clients. Updates are then processed and forwarded along the chain, until the Tail is reached. Query-requests (Q) from clients are accepted by the Tail. This way, only the Tail processes both Queries and Updates. Finally, for each processed request (Query or Update) the Tail generates a reply (R) to the appropriate client.



### 3.1.1 Queries are Processed by the Tail

The first thing to notice about the Chain Replication protocol is that Queries are processed by the Tail only. This is so for very a good reason. It ensures that the effects of an Update is reflected by Queries, if and only if that Update has been successfully processed by all replicas in the chain. Or, conversely, Updates that are only partially replicated will not be reflected by Queries.

If this was not the case, the consistency of the system would be much weakened. For example, consider an alternative protocol where all nodes in the chain accept and process Queries. Then it would be possible for a Query to reflect the effects of an Update U that was, say, only processed by the Head (and not by the Tail). Imagine then that all nodes except the Tail fail simultaneously. This may cause Update U to be lost somewhere in the chain, and never reach the Tail. As a consequence, subsequent Queries to the chain (now length 1) will not reflect the effects of Update U. To the client this would look like the effects of Update U vanished spontaneously. This is exactly the kind of behaviour one would like to avoid in a highly reliable storage system.

For this reason, in the original Chain Replication Protocol, Query processing is restricted to the Tail. In addition, Updates are not committed by the system until they have been successfully processed by the Tail. As illustrated by the above example, an Update that is partially replicated and then lost due to node failures, is indistinguishable from an Update that was never received by the system in the first place (given that Queries are processed by the Tail only). Thus, when we talk about the *most recent Update* to the system, this refers precisely to the most recent Update processed by the Tail.

On the other hand, restricting Query processing to the Tail is not without a cost. First, limiting Query processing to a single node, as opposed to all nodes, most likely reduces

Query throughput and increases latency. This might be problematic for Query-intensive applications. In addition, the load-balancing across the chain is not optimal. Especially, the Tail processes both Updates and Queries, whereas the other nodes only process Updates. This might cause the Tail to become a significant bottleneck for the throughput of the system. Furthermore, if the chain has buffer capacity for Updates, these buffers will fill up, thus causing the Update latency to become even higher.

Fortunately, there are ways to get around these problems. The idea is that in a real storage system all the data should not be exposed through a single chain. Rather, the data should be partitioned, with one chain responsible for each partition. Furthermore, when distributing these chains over a pool of servers, each server may be the host of multiple nodes, from different chains. For example, one server may serve the Head-node of chain A, the second node of chain B and the Tail-node of chain C. This way it would be possible to implement load-balancing between servers, even though the load-balancing across each chain is weak. In such a system, many problems associated with throughput, latency and load-balancing may be addressed by manipulating parameters such as partition-granularity, server-pool-size and chain-nodes per server.

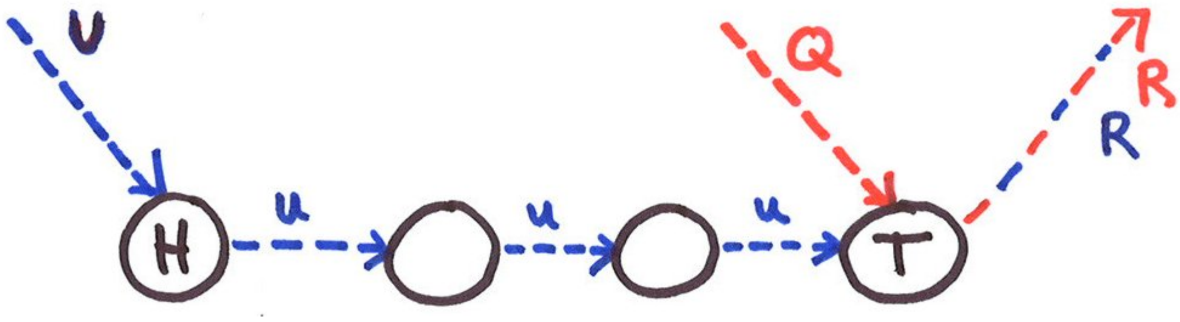
### 3.1.2 A Streaming Protocol

The second thing to notice is that Chain Replication is a streaming protocol. The Head accepts Updates from multiple clients, possibly concurrently. Still, update processing is done serially, thereby creating a single, ordered stream of Updates. This ordering is respected by all nodes in the chain and by the interconnecting links, which are required to be reliable, FIFO links. This FIFO property of the Update-stream is important for several reasons.

- First, it ensures consistency between replicas. All Updates are processed by all replicas in the same order. Chain Replication is consequently an instance of the State Machine Approach [FBS].
- Second, it allows for pipelining. A sequential strategy for Update dissemination generally implies higher latency. However, it is still possible to achieve high throughput, if Updates can be pipelined efficiently. Pipelining applications generally depend on the FIFO property of the pipeline.
- Third, ordered Update processing is essential for handling failures in the chain. This will be discussed further below.

Queries too are processed serially by the system, at the Tail. This means that there are two request streams running through the system, the Update-stream and the Query-stream. These two streams though, are not entirely independent. At the Tail, the Update-stream and the Query-stream are merged into one single, totally ordered request stream. This is essential because it means that the system implements *strong consistency*. All requests are processed serially at a single node, the Tail. It follows from this that a Query will always reflect the most recent Update to the system. Remember, from the above discussion, the most recent Update to the system is precisely the most recent Update processed by the Tail.

This figure illustrates the two streams through the chain, and how they are merged by the Tail.



### 3.1.3 Queries, Updates and Replies

In chain replication, a Query retrieves a copy of an object, or only a part of that object. Queries include an object identifier and possibly some options determining a specific part of the object. Thus, Queries are small messages and Query processing is typically i/o-bound rather than cpu-intensive.

Query-replies carry the data results of Query operations. Unlike Queries, Query-replies may potentially be sizeable messages. This depends on the object sizes in the system and the types of Queries supported. This however, is very much application dependent.

An Update is modeled as an operation applied to a single object. Updates carry an object identifier and some options. Typically options include an operation identifier and some associated parameters. The operation identifier refers to an application specific list of predefined operations. The implementation of an operation may refer to the given parameters and also the current value of the object (i.e. the object value - before the operation was applied). With respect to message size, parameters associated with operations may contain much data.

The chain replication protocol, as described above, indicates that Updates are forwarded and processed at each node in the chain. This is known as active replication [F.B.S]. However, there is an alternative way of implementing Update propagation in the chain. The Update operation could be processed only at the Head, and then the effects of that Update (i.e. the new object value) forwarded, rather than the Update itself. Even better, if Update effects are small (relative to object size) and easily described, it would be more attractive to forward only descriptions of differences between old and new values. This is typically an optimization with regard to bandwidth consumption. Depending on the application either strategy could be more efficient. However, there might be an additional reason to avoid active replication. Active replication does not support non-deterministic operations. In contrast, if only effects of Updates are forwarded along the chain, all replicas will be the same even if the operation (processed by Head) was non-deterministic. For this reason, the protocol as defined in [CR] forwards value differences instead of doing active replication.

Update-replies are typically small messages. If the Update operation yields a return value this is included in the Update-reply. Still, the main purpose of the Update-reply is to notify the client that the Update has been safely committed by the system.

## 3.2 Failures in the Chain

The serial architecture of the Chain Replication protocol is a good basis for implementing fault-tolerance and recovery after node failures. We explain the three basic failure scenarios of the chain. The Head node might fail, the Tail node might fail, or some intermediate node might fail. The problem of chain recovery after such failures is slightly similar to delete operations on a linked list. However, before we do that we state assumptions with regard to failures, and explain how node failures may cause messages to be lost by the protocol.

### 3.2.1 Assumptions and Failure Model

These are the theoretical assumptions.

- All chain nodes in the chain are assumed to be fail-stop processes. This means that failing processes immediately stop executing after a failure. In addition this state can be detected by other nodes.
- Network failures are modeled as process failures. For communicating processes, failures of the interconnecting network is indistinguishable from process failures.
- For a chain of N nodes, at most N-1 nodes may fail at once.
- Communication links between nodes are reliable and FIFO.
- No network partitioning.

In addition, with respect to implementation it may be reasonable to make the following practical assumption.

- Chain nodes are assumed to run in LAN environment. Interconnecting links have high bandwidth and low latency.

### 3.2.2 Lost Queries, Updates and Replies

Failing nodes may cause Queries, Updates and Replies to be lost. To see how this might occur, consider the following scenarios.

- Queries may be lost in transfer between the client and the Tail. This is possible even if the communication links are reliable. For instance, the Tail may fail after the Query has been successfully received by the network layer, but before it was processed by the protocol. Similarly, Updates can be lost in transfer between the client and the Head.
- Partially replicated Updates can be lost too. A partially replicated Update has been processed by a least one node in the chain, but has not yet been processed by the Tail. Such an Update can be lost if all nodes that have processed or received the Update fail at once.
- Replies may be lost in transfer between the Tail and the client. For instance, the Tail may fail after having processed an Update, but before it has been able to send the corresponding reply to the client. At this point, the Update will be successfully replicated at all live nodes, yet the client did not receive a reply.

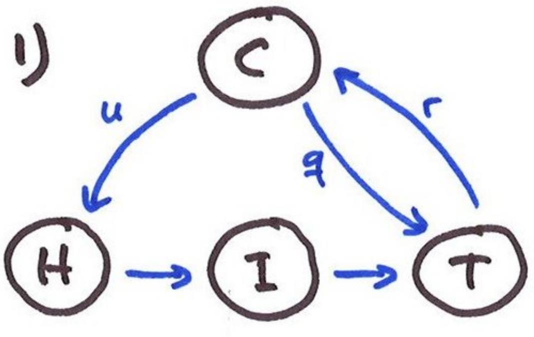
## 3.3 Head Failure

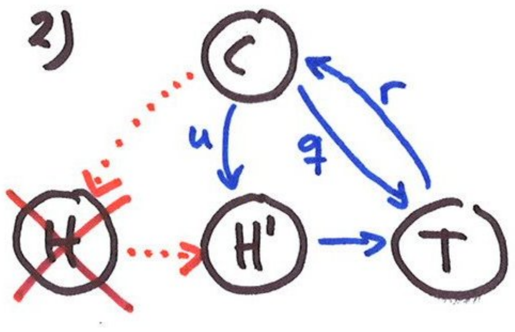
Head failure is the simplest type of failure. The failing Head may have processed some Update, which have not yet been forwarded to the successor node in the chain. If the Head fails, this Update will be lost (i.e. it will never be forwarded to the Tail). However, this is not really a problem. As pointed out above, Updates are not visible to clients until they have been processed by the Tail. So, for the client, Updates lost due to Head failures are indistinguishable from Updates lost in transfer between the client and the Head. The client will deal with both cases in the same way. For instance, it could re-issues the Update after a timeout.

Recovery after a Head failure will include the following steps.

- Remove the failed node from the chain.
- Instruct the successor of the failed Head to become the new Head of the chain.
- Set up a new connection between the client of the new Head

This transition is illustrated by figure 1) and 2). The client (C) is not part of the chain, but is affected during recovery.





Due to the assumption that only N-1 nodes may fail at once, Head failures can only occur in chains of length 2 or more.

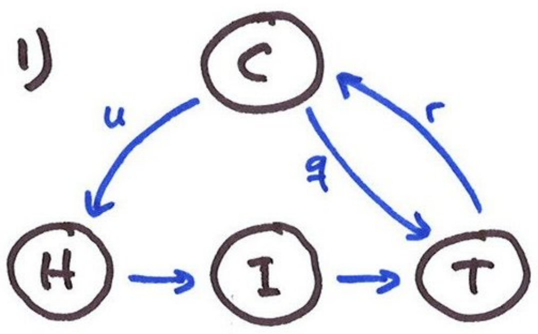
## 3.4 Tail Failure

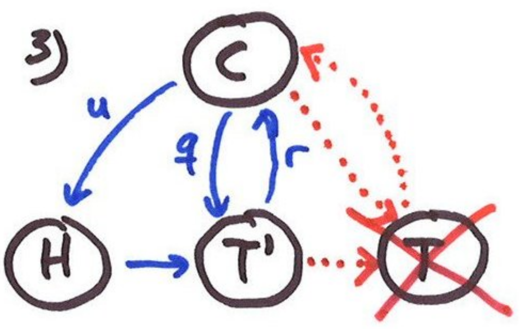
Tail failures are also fairly easy to cope with. The idea is that the predecessor of the Tail must become the new Tail. With regard to Update processing this transition is unproblematic. All Updates processed by the failing Tail have also been processed by its predecessor, so no Updates will be lost and the integrity of the chain is secured. However it is possible that Queries or Replies are lost in this transition. Again, this is not a serious problem. The client will notice the non-arrival of Replies and re-issue the corresponding Request.

Recovery after a Tail failure will include the following steps.

- Remove the failed node from the chain.
- Instruct the predecessor of the failed Tail to become the new Head of the chain.
- Set up a new connection between the client and the new Tail.

This transition is illustrated by figure 1) and 3). The client (C) is not part of the chain, but is affected during recovery.





Due to the assumption that only N-1 nodes may fail at once, Tail failures can only occur in chains of length 2 or more.

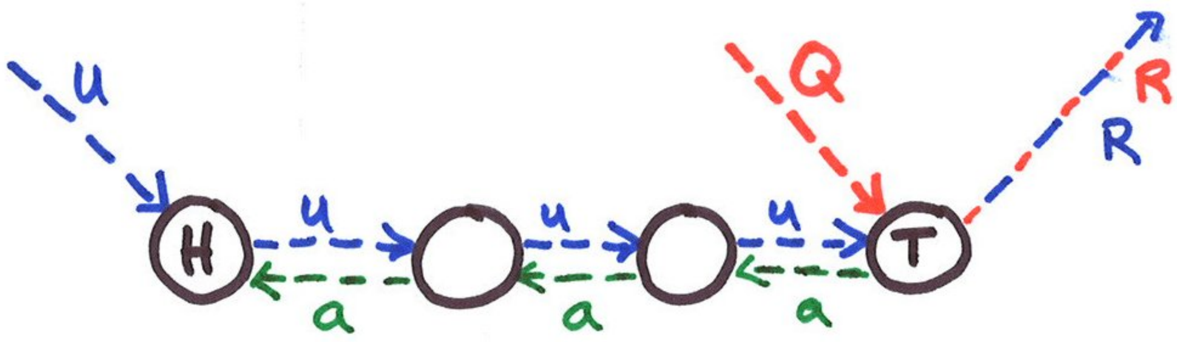
## 3.5 Intermediate Failure

Failures of intermediate nodes are simple in the sense that they do not involve the client at all. In fact, intermediate node failures are transparent to clients. The idea is that the predecessor and the successor of the failing node should become new neighbours in the chain. The challenge is to do this transition without loosing any Updates.

Consider the following scenario. The Intermediate has received an Update from its predecessor, yet it fails before the Update was forwarded and received by its successor. Now, simply removing the failed Intermediate from the chain would cause that Update to be lost (and replicas to be inconsistent). Instead, the predecessor needs to re-forward this Update to its new successor. In fact, the predecessor needs to re-forward *all* Updates that might not have been received by its new successor. However, this poses a new problem. How can the predecessor know which Updates have been received by its predecessor, and which have not?

The solution to this problem is to add acknowledgements (Acks) to the protocol. When an Update have been processed by the Tail, an Ack is generated and forwarded back up the chain. All nodes in the chain collaborate in forwarding these Acks until they reach the Head, their final destination. By looking at these Acks it is possible for each node in the chain to know which Updates have been successfully committed by the Tail.

The figure illustrates the Chain Replication Protocol extended with Acknowledgements (A).



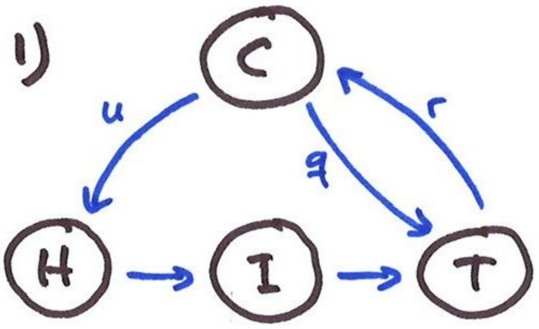
In the event of an Intermediate node failure, this mechanism can be used to figure out what Updates need to be re-forwarded (from the predecessor to its new successor). Here is how it works. Each node buffers Updates that have been successfully forwarded. We call this the Sent-buffer. Whenever an Ack is received, the corresponding Update is removed from the Sent-buffer. Then, after an Intermediate node failure, the predecessor re-forwards all Updates from the Sent-buffer to its new successor (in the correct ordering).

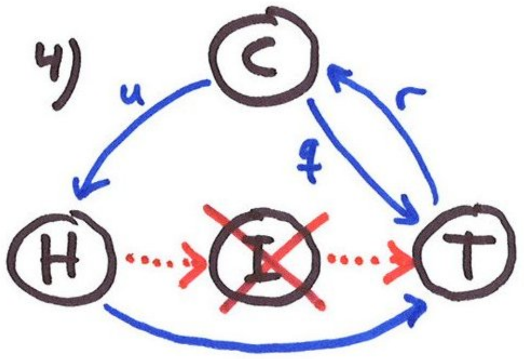
This solution ensures that no Update is lost. Still, it is not optimal in the sense that some Updates may be re-forwarded unnecessarily. As a consequence, the nodes must be able to detect and drop duplicate Updates. Fortunately, this is easy. For instance, the Head of the chain may associate sequence numbers with incoming Updates. These numbers can then be used by all nodes in the chain to detect duplicate Updates.

Recovery after a Intermediate node failure will include the following steps.

- Remove the failed node from the chain.
- Set up a new connection between the predecessor of the failed node, and its new successor.
- Re-forward all Updates from the Sent-buffer.

This transition is illustrated by figure 1) and 4). The client (C) is not affected during recovery.





Intermediate node failures can only occur in chains of length 3 or more.

## 3.6 Chain Extension

As we have seen, recovery from node failures involve reconfiguring the chain and reconnecting the client. However, this is not enough. In order to maintain the Chain length over time, a mechanism is needed that allows the chain to grow. We call this chain extension.

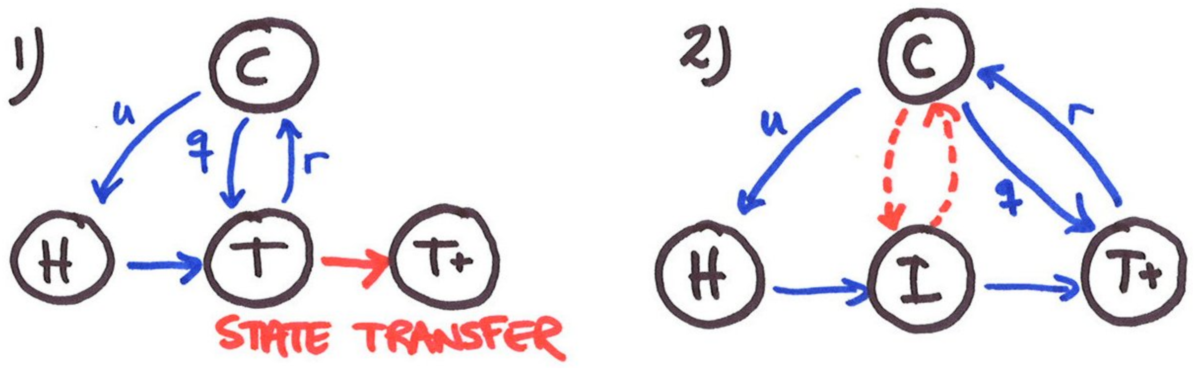
The idea is to safely add a new node to the chain, while the chain is still running. In principle, a new node could be inserted at any position in the chain. However, in the ODSI paper [CR] it is argued that appending to the Tail seems to be the most practical solution. Still, this is not trivial. Logically, the extension protocol has two steps. First, all the state that is replicated along the chain needs to be transfered to the new node. We call this state transfer. Then, as the state transfer completes, the responsibility of being the Tail can be passed over, from the old Tail to the new Tail.

The trivial solution would be to block Update processing in the chain while the state transfer is performed. This way, the state transfer could terminate in one pass, for instance by streaming a low-level disk image from the old Tail to the new Tail. However, blocking update processing in the chain, potentially for a long time is not a good idea, considering that high availability is the overall design goal for the system. Instead, a proper implementation will execute state transfer and update processing concurrently. Such a solution is more complex. Especially, new Updates may invalidate parts of the ongoing state transfer, requiring affected objects to be re-transfered. Thus, to guarantee termination of the state transfer, it may be necessary to block update processing for a little while after all. Still, this window of inactivity would be small and bounded. This way, chain extension can in practice be transparent to clients.

Such and implementation of chain extension will include the following steps.

- Connect new Tail T+ with current Tail T.
- Starts state transfer from T to T+.
- As state transfer comes close to completion on T:
    - Block Update processing on T.
    - Terminate state transfer on T.
    - Notify T that it is no longer the Tail. This implies disconnecting T from client.
    - Notify T+ that it is the new Tail. This implies connecting T+ and the client.
    - Unblock Update processing on T.

The following figure illustrates chain extension. 1) shows the situation while state transfer is taking place. 2) shows the new chain after Tail-responsibility has been passed to T+.



Chain extension needs to be fault-tolerant too. The transfer of Tail-responsibility from T to T+ is the commit-point of this protocol. If the new Tail T+ fails before commit, say during state transfer, the chain extension protocol needs to be aborted. If it fails after commit, this is equivalent with Tail-failure. If the current Tail T fails before commit, this too is equivalent to Tail failure. In this case, T+ never becomes a part of the chain. Finally, if current Tail T fails after commit, this is equivalent with Intermediate failure.

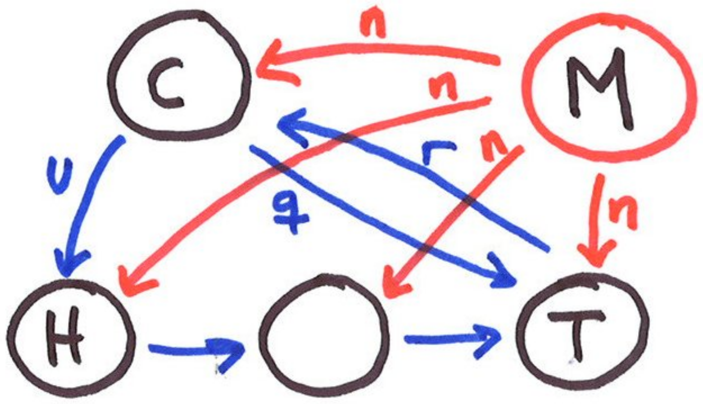
Note also that chain extension also solves the problem of chain setup. For instance, if you want to set up a chain of length N you do this by starting a chain of length 1, and then grow the chain until the desired length is reached.

## 3.7 Failure Detection and Recovery

In the OSDI paper [CR] failure detection and recovery is orchestrated by a replicated master process, external to the chain. The master process detects all failures and initiates recovery of the chain. This includes:

- Notifying chain nodes of the roles they need to play in the chain.
- Notifying chain nodes of new neighbours in the chain.
- Enable chain nodes to connect with new neighbours in the chain, after Intermediate failures.
- Enable the chain client to reconnect with the chain after Head or Tail failures.

The following figure illustrates how the master detects failures and notifies all relevant parties. Notifications from the master cause nodes to assume different roles in the chain, and to connect with new neighbours.



It is important to note that this master process, as presented in the paper, is a tool for explanation and proof, rather than a guide for implementation. It is even possible to do without one, provided that the functionality of the master is distributed reliably across the nodes of the chain. We have implemented such a solution, yet this is not the focus of this thesis.

## 3.8 Summary

The goal of achieving scale, high availability (fault-tolerance) and throughput, without compromising strong consistency guarantees, is the motivation for the Chain Replication protocol.

**Strong consistency** comes from the fact that all requests are totally ordered at a single machine, the Tail, and that the effects of updates are not visible to clients until they are processed by the Tail.

**High availability** is due to the simplicity of chain recovery. Especially, due to the strict linear topology, no time-consuming distributed coordination protocol is necessary for repairing the chain.

**High throughput** comes from the pipelining nature of the protocol. Especially, the query stream is never halted to wait for the completion of certain updates.

**Scalability** is in a sense external to the protocol. By having multiple chains run within a large pool of storage nodes, the throughput, the data volume and the supported number of clients may be scaled up. In addition, this allows load-balancing across storage nodes rather than load-balancing within a single chain.

These claims are backed by simulations in the OSDI paper [CR].

# 4. The Chain Client

An application may store application-objects reliably using chain replication. To do so the application must interact directly with a chain, thus becoming a client of that chain. Although the chain nodes may be assumed to run in a LAN environment, this assumption is not extended to clients in general. In fact, application clients may well communicate with the chain over a WAN.

## 4.1 System Interfaces

We define the chain client and the chain proxy, by presenting the central interfaces in a chain replicated storage system.

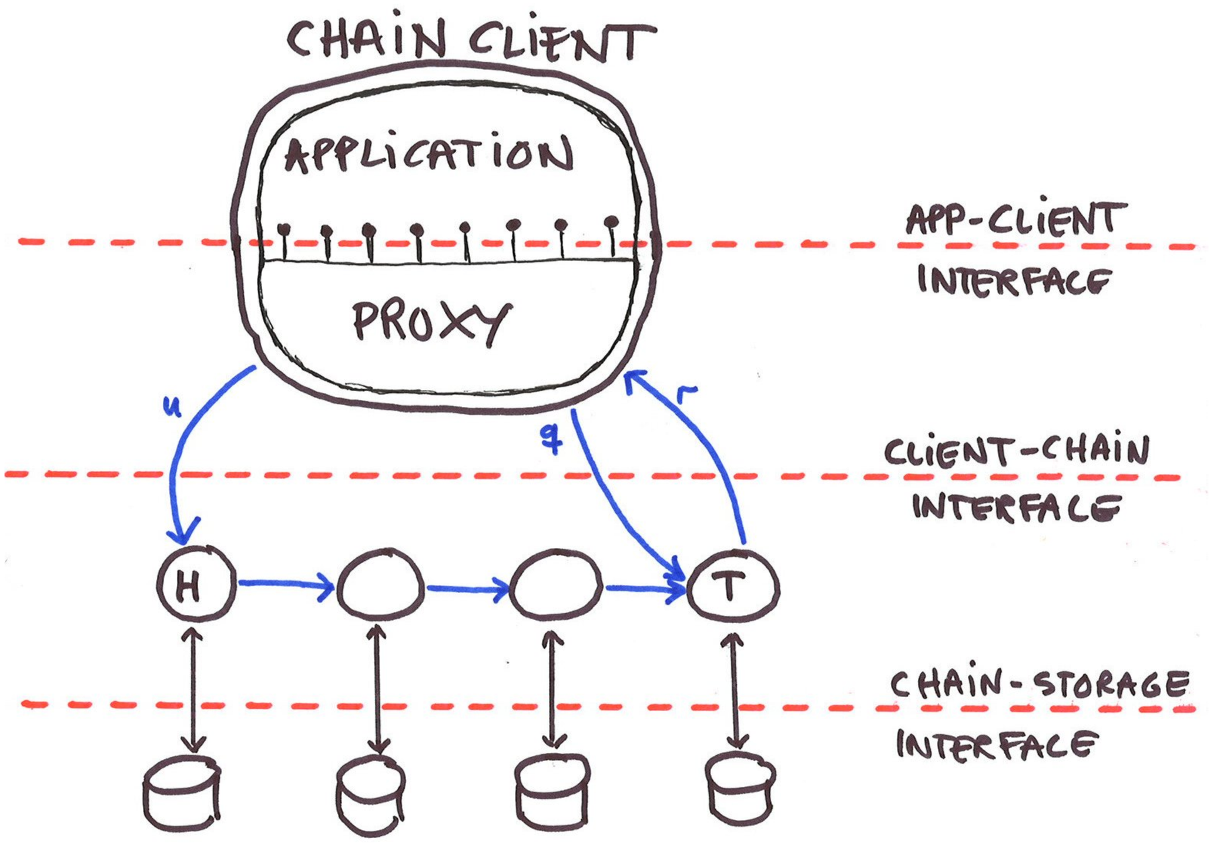
### 4.1.1 Client-Chain Interface

Chain clients all face the same challenges. They need to communicate independently with the Head and the Tail of the chain. Updates must be sent to the Head and Queries to the Tail. Replies are received asynchronously from the Tail. Furthermore, in response to failures in the chain, the chain client may have to switch seamlessly to a new Head or Tail. This implies that the chain client must receive notifications concerning changes of the chain configuration, and act appropriately in response to these notifications. All these challenges are defined by what we call the *client-chain interface*.

In addition, recall our objective of removing the dependency between the client and the master. Or, to be more precise, we want certain responsibilities of the master to be migrated to the chain itself. This implies that the client-master interface will be included in our client-chain interface. The details of how this is done will be discussed in the next section.

### 4.1.2 Application-Client Interface (Chain API)

As has already been indicated, implementing a chain client on top of the client-chain interface is not trivial. Fortunately, much of this complexity can be hidden from application programmers. The idea is to encapsulate generic client functionality within a *chain proxy*, and provide the application programmers with a higher-level API to the chain. We call this the *application-client interface*, or the *chain API* for short. The design of this chain API and the chain proxy that implements it, is the main focus of this thesis.



The figure above illustrates how the chain proxy fits in with the system as a whole. The chain client is comprised of two parts, the application and the chain proxy. The application-part is the actual application making use of the chain. The proxy implements generic client functionality bridging the gap between the application-client-interface (chain API) and the client-chain-interface. From the point of view of the chain, the proxy is the chain client. Conversely, from the point of view of the the application programmer, the proxy is the chain.

### 4.1.3 Chain-Storage Interface

The figure additionally introduces the *chain-storage interface*. This interface defines how chain nodes interact with their local storage service. The chain-storage interface essentially offers only two methods.[3]

**Storage API**
- queryReply = **Query**(queryRequest)
- updateReply = **Update**(updateRequest)

Update() invokes an update operation on the local storage object. The updateRequest given as parameter is simply a byte array, as far as the chain node is concerned. That goes for the updateReply as well. The Tail of the chain will include this updateReply in a Reply message and send it back to the chain client. The actual structuring of requests

3. Actually, in order to allow the chain to safely perform state-transfer during chain extension, there will have to be additional requirements to the chain-storage interface. Presenting those requirements is not within the scope of this thesis.

and replies is defined by the implementation of the storage service. The local storage service will parse incoming requests and serialise reply values according to some advertised protocol. We call this the *Storage Protocol*. Storage services can for instance be implemented on top of the local filesystem or a database.

This design is fortunate because it allows both the chain and the chain proxy to be implemented without reference to what objects are actually being replicated, or what kind of operations are permitted on these objects. In this perspective, the chain proxy and the chain together can be understood as a distributed routing mechanism, transporting request-byte-arrays from applications to replicas, and reply-byte-arrays in the opposite direction. The chain proxy accepts queryRequests and updateRequests from the application, and then delivers queryReplies and updateReplies as they become available. In fact, the chain API will even mimic the chain-storage interface by implementing the same methods. Thus, only the chain application need to be aware of the storage protocol.

**Chain API**

- queryReply = **Query**(queryRequest)
- updateReply = **Update**(updateRequest)

This makes it clear that the chain proxy and the chain together constitute a software layer that to some extent hides replication. At least in principle, this layer could be replaced by other instances of replication-middleware for the purpose of comparison.

## 4.2 The Chain Client is a single Process

Note also that the above illustration includes both application and chain proxy within a single logical chain client. What this means is that application and proxy should run in the same process, or at least on the same physical machine. We may think of the proxy as running in a thread within the application process, or possibly as a backend deamon. Especially, what we want to avoid is for the proxy to run independently on a separate machine. The reason is that this would introduce a new type of failure into the system. The Proxy may fail when the application does not. Such failures will hurt the availability of the replicated data, and it would force us to devise a way of replicating the proxy as well. Instead of doing that, it is better to simply require the proxy to be running as a part of the client application.

## 4.3 The Chain API

In order to understand how the chain proxy will work, it is helpful to start by taking the perspective of the application programmer. In this section we present a rough overview of some important features of the chain proxy, and how these features affect the design of the chain API. The details of the chain API will be discussed further in section 7.

As mentioned above, the chain API is comprised essentially of two primitives; Query() and Update(). However, the pipelining nature of chain replication adds slightly to the complexity by requiring these primitives to be non-blocking (asynchronous) send-operations. So, the basic API will consist of a few non-blocking methods and corresponding asynchronous callbacks.

- queryID = **QueryAsynch** (objectID)
- updateID = **UpdateAsynch** (objectID, OpType, Params...)

- **SetQueryReplyHandler**(QueryReplyHandler)
- **SetUpdateReplyHandler**(UpdateReplyHandler)

- **QueryReplyHandler**(queryID, reply) (...)
- **UpdateReplyHandler**(updateID) (...)

To use this API, the programmer needs to implement handlers for incoming replies and register them with the client. Then s/he is all set to send Queries and Updates to the system. The Updates will be transferred to the Head of the chain, in the order received. This ordering is maintained throughout the chain, so Update-replies will be received in the same order. There is no reply for lost Updates. Similarly, all the same is true for Queries, except they are transferred to the Tail, not to the Head.

### 4.3.1 Ordering of Queries and Updates

It is important to note that ordering is not preserved *between* Queries and Updates. The chain proxy will manage outgoing Queries and Updates as two completely independent streams. For example, if an application issues a Query followed by an Update, the proxy may well send the Update to the chain, before sending the Query. There are good reasons for this. First, the independence between Queries and Updates is a great feature of the protocol. It allows Query processing to continue even when there are failures in the chain. Conversely, if the Tail fails and halts the Query processing, Update processing will still be able to continue at the remaining nodes. The chain proxy must allow applications to take advantage of this feature. Second, and more importantly, even if the chain proxy did preserve the ordering between issued Queries and Updates, the chain would not. From the point of view of the chain, the correct ordering of a Query and an Update is defined by the order in which they are processed by the Tail, not the order in which they were issued by the chain proxy or the application.

So, if an application issues an Update U immediately followed by a Query Q, for the same object, there is no guarantee that Query Q will reflect the effects of Update U. In fact, it is rather likely that it will not, because Update U must traverse the chain while Query Q goes directly to the Tail. This may be puzzling, considering that the chain is supposed to implement strong consistency. The source of confusion here is that sending an Update is a non-blocking operation. If the application wants Query Q to reflect the effects of Update U, it must delay Query Q until the Reply associated with U is received.

### 4.3.2 Lost Queries and Updates

Earlier, in section 3, we have discussed how Queries, Updates and Replies can be lost by the chain. For client applications, learning about such losses may be crucial. The way this is done in chain replication is not different from most client-server architectures. If the proxy does not receive a reply for a given request, within some bounded time interval (Timeout), this means that the request is lost. So, the proxy must associate a timestamp with every outgoing request and notify the application if the corresponding reply was not received in due time. The application may then choose to retransmit the request, or take other appropriate actions. Application programmers specify this by implementing timeout-handlers. These handlers will be invoked by the proxy in the event of a timeout. Thus, the above API needs to be extended with the following methods.

- **QueryTimeoutHandler**(queryID) (...)
- **UpdateTimeoutHandler**(updateID) (...)
- **SetQueryTimeoutHandler**(QueryTimeoutHandler)
- **SetUpdateTimeoutHandler**(UpdateTimeoutHandler)

Although the chain proxy deals with lost requests in a traditional way, two protocol specific features are worth mentioning. First, Query-latency and Update-latency are not the same in chain replication, so it may be a good idea to manage separate Timeout values for Queries and Updates. Especially, Update-latency depends on chain-length, so this might have to be modified as a response to chain reconfiguration. Second, it may sometimes be possible to conclude that a request is lost, before the Timeout has expired. For example, if Update-reply #44 is received after Update-reply #42, the chain proxy can conclude that Update #43 is lost. This is because the Update-stream (and the Query-stream) are ordered streams.

Unfortunately, like any client-server architecture, a mechanism for detecting lost requests might also imply false negatives. For example, a reply might be lost in transfer between the Tail and the chain proxy. This would be the case if the Tail fails after having processed the request, but before having sent the reply. This means that whenever a request is lost, the application client can not know whether the request was processed by the chain, or not. Especially, with regard to Updates, this might be a problem. The application may resolve this situation in two ways. If the Update operation is idempotent, it is safe to just reissue the Update immediately. Alternatively, for non-idempotent operations, it would be necessary to query the chain for the effects of the Update. If the effects are not found, the Update was lost and can safely be reissued. Or, if the effects are found, all is fine. Still, this approach might be time-consuming, or problematic for other reasons, so the application programmers might want to avoid non-idempotent Update operations, if possible.

# 5. The Session Protocol

In section 4 we discussed the application-client interface (chain API) and the client-chain interface. Together, these two interfaces define the chain proxy. So, at this point we should be ready to present a design for the chain proxy. However, before we can do so we need to discuss how the responsibilities of the master can safely be migrated to the chain. This section presents our solution to this problem, the session protocol.

## 5.1 Removing the Client-Master Dependency

Remember that after a chain failure the chain clients need to be notified about changes in the chain configuration. Essentially, if there is a new Head or a new Tail in the chain, the chain clients need to know their addresses in order to reconnect and resume operation. In addition, chain clients need to know about changes in chain length in order to adjust timeout values for Updates. For this reason Notifications carry the following information concerning the configuration of a chain at a given point in time.

> *Notification : <Head-IP, Tail-IP, ChainLength>*

Upon receipt, chain clients will compare the Notification to its current view of the chain, and take the appropriate actions.

Now, an implementation of the original client-master-interface need not be difficult. For example, the chain proxy will connect to the master in order to start a client-session. After that the master will send a notification to the chain client (on that connection) whenever a relevant change occurs in the chain configuration. At some point the application client will terminate the session simply by closing the connection.

Still, we do not want to do this. Instead we want to avoid altogether this dependency between the chain proxy and the master process.

> *The idea is to migrate the responsibility for notifying chain clients (of chain reconfiguration events) from the master to the chain itself.*

There are two reasons why this is an attractive idea. First, it simplifies the system architecture. Chain clients can interface with the chain only. No knowledge of the master process is required. Second, it is the first step towards removing the master entirely from the system. If failure-detection and chain-recovery can be implemented reliably within the chain itself, the master process becomes obsolete. This would make the system architecture both simple and elegant. This though, is not within the scope of this thesis.

Next we present our solution to this problem, the session protocol. In fact, the session protocol consists of two protocols, the *session-start protocol* and the *session-stop protocol*.

## 5.2 The Session-Start Protocol

The master has two responsibilities with respect to chain clients. First, it reliably stores information about all clients of the chain. This could for instance be a list of IP-addresses. Such a list enables the master to notify all clients after some change has occurred in the chain configuration. Second, this list helps define a *client-session* in the system. A chain client maintains a session with a chain, as long as its IP-address is included in the client-list of the master. So, to start a client-session the chain client must connect and register with the master. To end a client-session the chain client unregisters with the master and disconnects. Note that this definition of client-session does not depend at all on the state of any connection between the chain client and the chain. This is a good thing, because both the Head and the Tail of a chain may well fail at once. If this happens, no client has any connection to the chain. Yet, because the liveness of their sessions are independent of this, they can all be notified as the chain recovers, and eventually resume their operations.

So, to eliminate the dependency between the chain client and the master, the following problems need to be solved within the chain.

1. How to store client information reliably in the chain?
2. How chain clients can start and end a session with the chain?
3. How the chain can notify the chain clients of configuration events?
4. How connections are established between Head, Tail and chain clients?

Fortunately, the answer to the first question is almost self evident. Client information can be stored reliably in the chain, replicated like any other data using the basic primitives of chain replication. Thus, each node of the chain has a replica of the entire client-list. Two Update operations will be associated with this object, Register and Unregister. Register adds a new entry to the list, including an IP-address and a port-number. Unregister removes one such entry. Note that this answers the second question too. Chain clients start a session by sending a RegisterUpdate to the Head of the chain, and end the session by sending a UnregisterUpdate. Both operations should be implemented as idempotent operations.

The chain also needs to notify all chain clients of chain configuration events. Because the client-list is now replicated along the chain, any node in the chain could do this job. Still, the Tail seems to be the best candidate, because it already sends Replies to all chain clients. It is easy to send Notifications using the same connection. However, if the Tail is going to notify the chain client, we still need to make sure the Tail learns about all the relevant chain events. For instance, the Tail needs to know that the chain has a new Head. From the earlier presentation of the protocol, it is not obvious that the Tail has this information. If the master always keeps global information about the chain configuration, each node in the chain only need to know about its immediate neighbors. So, to make this work, we need to assume that the master feeds the Tail especially with all Notifications (not only those relevant to the Tail). In effect, rather then having the master broadcast Notifications to the chain clients directly, the master sends one Notification to the Tail, which now does the broadcast instead.

Finally we may discuss how connections are created between the chain client and the chain. There are two connections. One connects the chain client with the Head. The other connects the chain client with the Tail. We call them Head-connection and Tail-connection, respectively. The Head-connection is used to transfer Updates, while the Tail-connection transfers Queries in one direction and Replies and Notifications in the other direction. Three situations require these connections to be created or re-created.

1. **Session-Start**. The chain client wants to start using the chain, and needs both connections in order to do so.
2. **New Tail**. The Tail connection has been lost due to Tail failure. A new Tail-connection is needed as soon as a the chain gets a new Tail.
3. **New Head**. The Head-connection has been lost due to Head failure. A new Head-connection is needed as soon as the chain gets a new Head.

We need to decide which party is responsible for creating connections in each of these cases, the chain client, the Head or the Tail.

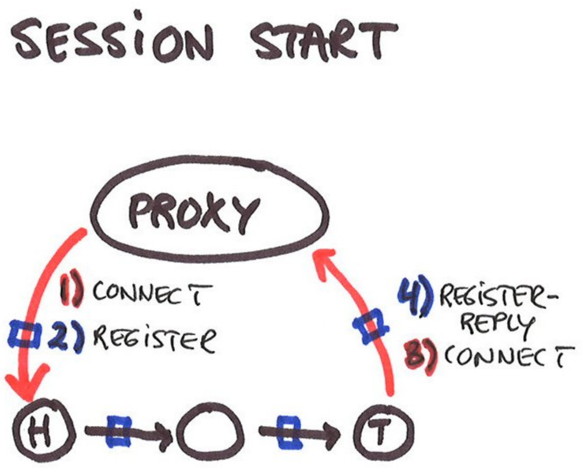
We start by considering the Head connection. The chain client starts a client-session by sending a RegisterUpdate to the Head. However, to do so it must first create the Head-connection. Thus, the chain client, not the Head of the chain, will be responsible for creating the Head-connection in this situation. This strategy also works when there is a new Head in the chain. The Tail will notify chain clients about the new Head, and the chain clients will then create the new Head-connections in response to this.

Next we discuss the Tail-connection. Above we described how the Tail of the chain is responsible for notifying chain clients of relevant chain events. So, when there is a new Tail in the chain this new Tail must immediately notify all chain clients of its existence. However, in order to do so, the new Tail must first create new connections to all those chain clients. Thus, in the Tail of the chain, not the chain client, will be responsible for creating the Tail-connection when there is a new Tail. This strategy also works for the session-start protocol. When the RegisterUpdate reaches the Tail of the chain, the Tail creates the connection to the appropriate chain client, and then immediately sends the RegisterUpdateReply.

This makes for a quite simple solution.

- *The chain client is always responsible for establishing the Head-connection.*
- *The Tail of the chain is always responsible for establishing the Tail-connection.*

Session-start now only requires chain clients to create the Head-connection and then send the RegisterUpdate. The Tail then creates the Tail-connection and sends back the RegisterUpdateReply. When the chain client receives this Reply, the system is ready to use. Note that the session-start protocol will be required of all chain clients, including those that do not intend to send any Updates. The session-start protocol is illustrated by the figure below.



Still, one small issue remains. During session-start, the chain client still needs to know the IP-address of the Head, in order to create the first Head-connection. This is similar to knowing the IP-address of the master, though failures in the chain implies that the IP-address of the Head is not static like the master. We assume that there is some service that can help chain clients locate the Head of a chain during bootstrap. A trivial solution would be to simply probe all nodes in the server pool, until the correct one is found.

In summary, the session-start protocol and the connection-recovery protocols work as follows.

**Session-Start**

1. Chain client creates Head-connection.
2. Chain client sends RegisterUpdate over the Head-connection.
3. After RegisterUpdate is processed by Tail, Tail creates Tail-connection and sends a RegisterUpdateReply.
4. Chain client may start using the chain upon receipt of RegisterUpdateReply.

**New Tail**

1. A node becomes the new Tail of the chain.
2. New Tail uses replicated client-list to create new Tail-connections to all chain clients.
3. New Tail sends Notifications of new-Tail event to all chain clients.

**New Head**

1. A node becomes the new Head of the chain.
2. Tail notifies all chain clients of new-Head event.
3. All chain clients create new Head-connection after receiving Notification.

## 5.3 The Session-Stop Protocol

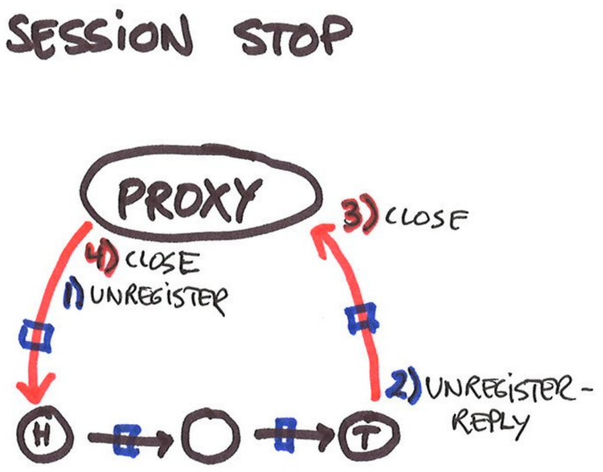
A client session can be terminated for four reasons.

1. The client application wants to terminate the session.
2. The chain client fails.
3. The chain wants to terminate the session.
4. The chain fails, i.e. all nodes in the chain fails simultaneously.

### 1. Client Application Terminates Session

When the client application wants to terminate the session, the Session-Stop protocol is analogous to the Session-Start protocol.

1. The client sends an UnregisterUpdate to the Head of the chain.
2. This request is then processed by the chain, causing the client to be safely removed from the replicated client-list.
3. The Tail generates an UnregisterUpdateReply and sends this over the Tail-connection.
4. The client receives the UnregisterUpdateReply and terminates both the Head-connection and the Tail-connection.



The Head-connection is not terminated until the UnregisterReply is received. This ensures that the connection is not taken down before the Head has received the request. Also, the connection might be useful if the RegisterUpdate is lost and needs to be retransmitted. To be precise, the Head-connection will only be useful again if the request is willfully dropped by the Head of the chain (before being processed). In contrast, if the request is lost due to node failures, a new Head-connection is required to a new Head, before the UnregisterUpdate can be retransmitted.

### 2. Client Application Fails

Failures of chain clients do not threaten either the integrity of the chain nor its replicated data. Still, it would be a good idea to try to clean-up after failed clients. Especially, it would be nice to be able to eventually remove failed clients from the replicated client-list. First, in order to do so, the chain needs to *detect* failed clients. Remember, the master can no longer be relied upon to do this failure detection, since it has no knowledge of clients in our revised system architecture. On the bright side though, the failure detection need only be best effort. The consequences of keeping a few dead clients in the client-list are not dire.

A practical solution would simply rely on the Head or the Tail being able to detect failures related to the Head-connection or Tail-connection, respectively. Next, after a client failure has been detected, an UnregisterUpdate needs to be injected at the Head, on behalf of the failed client. This will cause the client to be safely removed from the replicated client-list. Finally, when the UnregisterUpdateReply reaches the Tail, the Tail needs to shut down the appropriate Tail-connection, rather than performing the usual action of generating a Reply and relying on the client to close the connection. For this reason, the UnregisterUpdate injected by the Head must be distinguishable from the UnregisterUpdate issued by a live client. The Head may use the payload of the UnregisterUpdate for this purpose.

One specific option would be to let the Tail be in charge of failure detection. There is however an added challenge with this solution. After having detected a failure, the Tail needs somehow to inject an UnregisterUpdate at the Head of the chain, thereby playing the role of chain client relative to the Head. So, a seemingly better solution would be to let the Head do both failure detection and injection of the UnregisterUpdate.

After forwarding the UnregisterUpdate, the Head may perform the necessary clean-up after the failed client.

In any case, failure detection can easily be implemented by both Head and Tail. A practical solution would be a UDP-based Ping-Pong protocol. The chain would send a Ping periodically to a well defined Port on all the chain clients. If the chain client has not responded to a certain number of Pings, the chain may conclude that the chain client has failed. Other signs of client liveness, such as received Requests or non-failed connections may also be included in such a protocol. So, in order to support failure detection, chain clients will be required to respond to UDP Ping messages .

UnregisterUpdates may be lost due to failures in the chain, just like any other Update. This is not a reason to worry though. If all nodes that have processed the UnregisterUpdate fail simultaneously, the remaining chain will eventually have a new Head. This new head, or the Tail (depending of the above choice), will then eventually (re-)detect the failure of the failed client (still present in the client-list) and inject a new UnregisterUpdate. In effect this implements retransmission of UnregisterUpdates.

**3. Chain Terminates Session**

One could also imagine that the chain took initiative to end a client session. Possible reasons for wanting to do so could be high load, temporarily reduced performance as a result of chain recovery or chain extension in progress, or simply a system shutdown. The solution for client failures 2) could be abused to to this. The chain could simply conclude that the client has failed (although it has not) and follow the steps outlined above. This would cause both the Head-connection and the Tail-connection to be shut down by the chain. Unfortunately, from the perspective of the client, this is indistinguishable from a scenario where Head and the Tail fails simultaneously. The client will not know that the session has been ended.

Admittedly, allowing the chain to shut down a client session is somewhat at odds with the overall goal of chain replication; High availability. Thus, the design of a chain Session-Stop protocol that safely informs the client, before shutting down the connections, can reasonably be kept outside the scope of this thesis. However, in a real system, such a mechanism might be practical.

**4. Chain Fails**

The simultaneous failure of all nodes in a chain violates the basic assumptions of the chain replication protocol. Still, in practice it may happen, due to short chains, or violations of other assumptions such as the independence of failures between nodes. In any case, the client will not know that a chain has failed. Again, from the perspective of the client, a complete chain failure is indistinguishable from the simultaneous failure of Head and Tail only. So, the client will keep waiting for notifications, forever. A simple solution to this problem would be to introduce a session timeout. If both the Head-connection and the Tail-connection is down, the client, after some reasonable time interval, may conclude that the chain has failed. Incidentally, this also solves problem 3). If the client session is ended by the chain, the client will eventually time out and conclude that the chain failed. This though, is not completely satisfying. There is a difference between having your session terminated, and having the chain fail completely. In the first case, your data is unavailable, yet safe. In the latter case your data is potentially left in an inconsistent state.

Our proxy design includes a chain timeout. The time will be measured from the point where both Head-connection and Tail-connection have failed. If there is not a new Tail-connection withing a bounded time interval, the proxy will conclude that the session ended. As part of cleanup it will time out all pending requests before it notifies the application that the session has ended.

## 5.4 Session API and Local Session View

The chain API will provide special methods SessionStart() and SessionStop() allowing a client session to be started and stopped. The SessionStart() operation demands the IP-address of the Head of the chain. Internally the chain proxy will keep a local view of the state of a session. The local view of the session state is made available by the SessionStatus() method. It returns three boolean values. The first value says whether the proxy is in a session or not. The two other values indicate whether either the start-session protocol or the stop-session protocol is in progress at the moment.

- **SessionStart**(headIP)
- **SessionStop**()
- (session, start, stop) = **SessionStatus**()

The chain proxy also defines upcall-handlers which are invoked on certain state transisions. Upcall handlers for the session protocols may be defined using the following setter methods. SessionStartHandler() will be invoked when the session-start protocol has completed, thus it indicates that the session is operational. SessionStopHandler() will be invoked when the session-stop protocol has completed. The SessionTimeoutHandler() will be invoked after the session has timed out.

- **SetSessionStartHandler**(SessionHandler)
- **SetSessionStopHandler**(SessionHandler)
- **SetSessionTimeoutHandler**(SessionHandler)

When the chain proxy is not in a session, it will not permit any operations to be performed, except SessionStart().

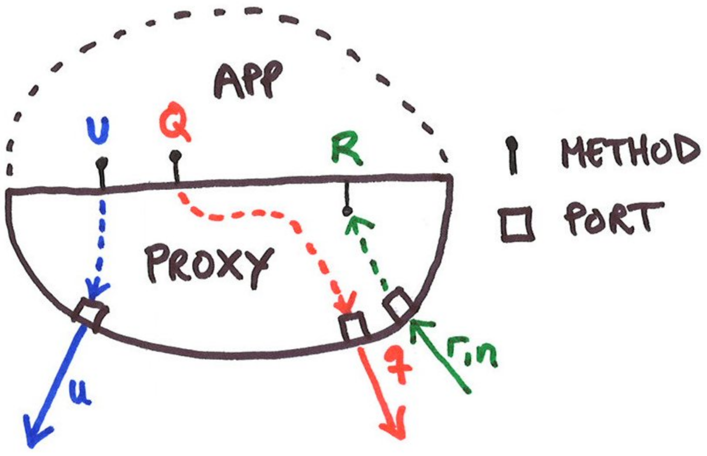
Finally, note that the utility of this session API is not really limited to our revised system architecture. Even if we did not remove the client-master dependency, the chain client would still need to start and stop a session. The only difference, apart from the implementation of this API, would be that SessionStart() would have to accept the master-IP as parameter, rather than the Head-IP.

# 6. Message Ports

Before we approach the design of the chain proxy, we first want to change the focus from low-level connections to higher-level message streams and ports. The abstraction of Message Ports is the basis for both the design and implementation of the chain proxy.

## 6.1 Message Streams

Chain replication is a streaming protocol for discrete messages. So, when we discuss the chain proxy, the message stream is indeed a helpful metaphor. Essentially, three message streams run through the chain proxy. The application (App) delivers Updates and Queries by invoking appropriate methods. The chain proxy writes Update-messages to the Head-connection and Query-messages to the Tail-connection, thereby keeping the two streams independent. The third stream runs in the opposite direction. Replies and Notifications arrive at the Tail-connection, and Replies are forwarded to the application. The Query-stream and the Reply/Notification-stream share the Tail-connection. The below figure illustrates these message streams running through the proxy.



## 6.2 Message Ports

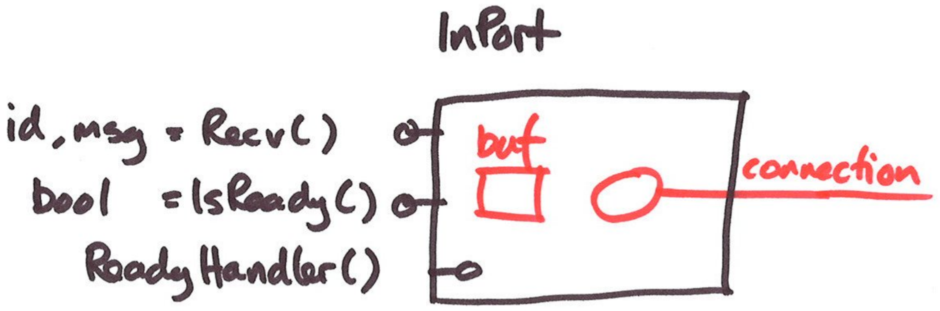
The figure also introduces message ports. Essentially, a port is a thin wrapper around a network endpoint, i.e. a socket. The port hides the complexity associated with reading and writing discrete messages to/from the network. *Out-Ports* accept a single message at a time and take responsibility for writing the message completely to the network before accepting a new one. Similarly, *In-Port*s read data from the network and delivers complete messages to the program. The naming is relative to the program using the ports. An In-Port transfers messages *in* from the network, whereas an Out-Port transfers messages *out* to the network. The chain proxy has four such ports, associated with its two connection. Head-Out-Port writes messages (Updates) to the Head-connection. Tail-Out-Port writes messages (Queries) to the Tail-connection. Tail-In-Port reads messages (Replies, Notifications and Pings) from the Tail-connection. Head-In-Port reads messages (Pings) from the Head-connection.

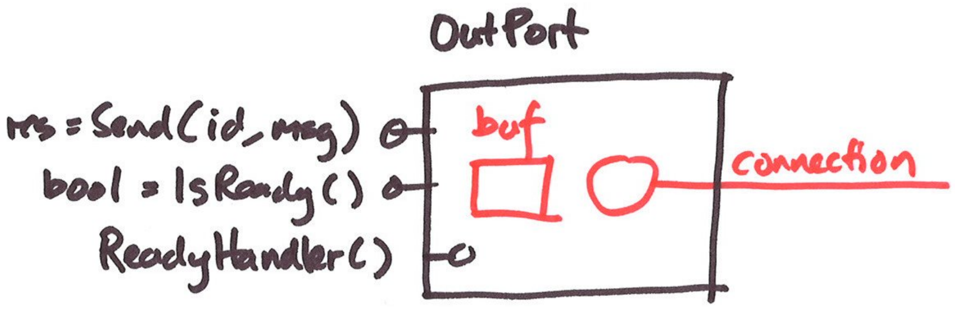
## 6.3 The State of a Message Port

Each message port may be in one of two possible states, ***Ready*** or ***Not-Ready***.

> **In-Port**: An In-Port is Ready if it has successfully read a complete message off the network. Thus, when an In-Port is Ready, the program is guaranteed not to block when invoking InPort.Recv(). As soon as the program has received the message, InPort will re-enter the Not-Ready-state, until a new message is completely read. The initial state of an InPort is Not-Ready. We may think of the In-Port as having a buffer size of 1 message. Recv() empties the buffer, thereby triggering continued reading from the network.
>
> **Out-Port**: An Out-Port is Ready if it is safe to send a new message. This implies that the previous message was completely written to the network buffer, or that there was no previous message. The initial state of an Out-Port is thus Ready. As soon as the program has invoked OutPort.Send(message), the OutPort enters the Not-Ready state, and stays there until the given message is completely written to the network. We may think of the Out-Port as having a buffer size of 1 message. The buffer is not emptied until the message is completely written to the network buffer.

Note that the ports have no specific state reflecting the condition of the underlying network connection, say for instance a Disconnected-state. Instead, the state of the network connection is covered by the Ready/Not-Ready states. For example, consider an Out-Port experiencing a network failure while writing a message. Because the Out-Port can never complete writing its message, it will remain forever in the Not-Ready state. Similarly, if an In-Port looses its network connection during a read, it too will remain forever in the Not-Ready state. Note also that the Out-Port may be in the Ready state although it might not be associated with an operational network connection. This is because it has buffer space of 1 message, regardless of the state of the underlying network. The below figure illustrates In-Ports and Out-Ports.





## 6.4 Replacing the Network Connection within a Message Port

A central challenge of the chain proxy is replacing failed connections. For example, when the Head of the chain fails, the Head-connection will fail with it. The chain proxy does not necessarily have to do anything particular in response to this event. However, at some point later the chain proxy will receive a Notification from the chain, with a reference to the new Head. Now the chain proxy must connect to the new Head, and somehow replace the old, failed Head-connection with the new one.

The port abstraction is valuable for this purpose as well. Both In-Ports and Out-Ports offer an additional primitive Port.SetConnection(newConnection) that simply replaces the underlying network connection with the given. This operation is safely integrated with the Ready/Not-Ready state transitions of the port.

> **In-Port**: A Ready InPort stores one message in its buffer. If the underlying network connection is being replaced in this state, not much will happen until the next invocation of InPort.Recv(). This will empty the buffer and allow the Port to start reading from the new connection. If the InPort is Not-Ready this means that a message has been read partially or not at all. Replacing the connection now implies deleting the possible remains of partial messages and start reading a new message from the new connection.
>
> **Out-Port**: If the Out-Port is Ready, this means that its size 1 message buffer is empty and that the previous message was written completely. The connection can be safely replaced. The next invocation of Out-Port.Send(message) will fill the message buffer and start writing to the new connection. If the Out-Port is Not-Ready, this means that the message buffer is full, and that the previous message was only partially written to the old (presumably failed) connection. When a new connection is provided the Out-Port starts writing the entire buffered message to the new connection.

## 6.5 Upcalls from a Message Port

Message ports exposes two types of internal events to the program, by invoking predefined upcalls. First, the event that the Port becomes Ready. This allows the program to take appropriate action immediately after the port has entered the Ready-state. Especially this presents a valuable alternative to repeatedly polling the state of the port. When first instantiated, an OutPort is Ready by default. This default state will is not signalled by such an upcall. The second event is that of a the network connection being disconnected or closed. This allows the programmer to take appropriate action, say replace the failed connection, as a direct response to the event. In addition, the ports are logging number of internal events conserning the reading and writing of messages. The logger output may be helpful for debugging and performance testing.

## 6.6 Port Modes

Finally, message ports offer two modes of execution, *Event-Based* and *Procedural*. These two modes affect the semantics of Recv() and Send() operations, together with their associated ReadyHandlers(). These port modes are motivated by application programmer demands as well as some implementational issues. Here follows a presentation of two particular problems with OutPorts. Port modes are then presented as our solution to these problems.

**Problem A: Upcalls and infinite method-nesting.**

As mentioned above, the motivation for implementing a ReadyHandler() is to be able to trigger application activity as a reaction to the NotReady -> Ready state transition within a port. This would be an alternative to polling for state changes. For instance, when an OutPort becomes Ready the application may want to send a new message. The following code snippet illustrates how we would like application programmers to express this.

```
def ReadyHandler(self, thisPort):
    self._msg_id += 1
    thisPort.Send(self._msg_id, "Message")
```

Now, consider what happens when this code snippet is executed. The Send() operation will try to write the message to the network. While doing this the port is NotReady. Thus, if the write operation succeeds directly the OutPort should re-enter the Ready state, invoke the ReadyHandler() and return. For instance, a simple implementation of the Send() operation could look like this.

```
def Send(self, id, msg):
    self.READY = False
    result = self.Write(id, msg)
    if result == "success":
        self.READY = True
        self.ReadyHandler()
    return result
```

Now, there is an obvious problem with this solution. Send() invokes ReadyHandler() and ReadyHandler() invokes Send(). This might potentially lead to infinite method-nesting. Even if it did not it could still be very confusing.

So, a solution is to modify Send() so that ReadyHandler() is invoked only strictly *after* Send() has returned. The details of how this is done is discussed in section 8. So, at this point it suffices to hint a solution. Recall that Ports accept responsibility for completing read and write operations later, if they cannot be completed immediately. The same mechanism that allows this may also be used to execute the ReadyHandler().

**Problem B: Repeated Invocations**.

Another thing that we would like ports to support are repeated invocations. For instance, as long as a Send() operations is immediately successful the natural thing for the programmer is to issue the next. Here is a snippet of application code that illustrates this scenario.

```
def SendRepeatedly(self, outPort, msgQueue):
        while outPort.IsReady():
                self._msg_id += 1
                outPort.Send(self._msg_id, msgQueue.pop())
```

Now, this seems like a reasonable thing to do, but there is a problem here as well. It conflicts with our previous solution for the method-nesting problem. If the ReadyHandler() is supposed to be invoked strictly *after* Send() returns, there is no simple way to ensure that it is invoked *before* the next Send(). As a consequence, the port is not guaranteed to be Ready when ReadyHandler() executes. This essentially makes the ReadyHandler() pointless.

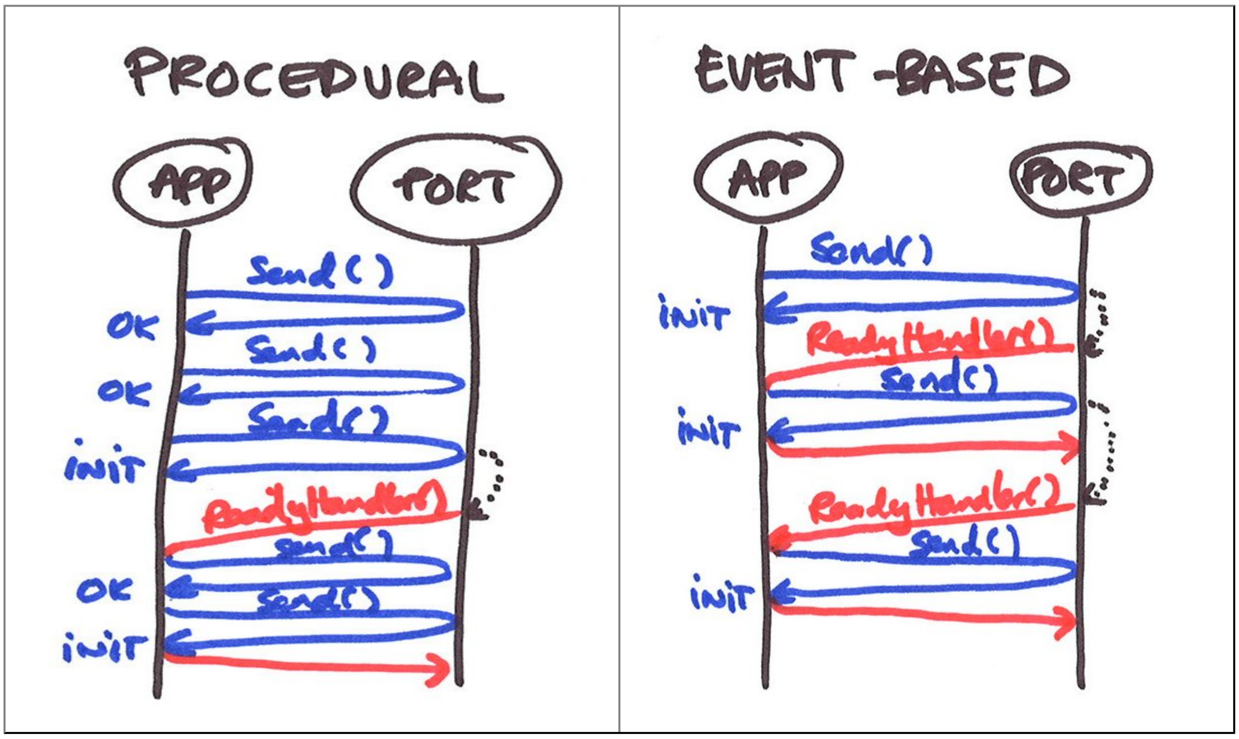
**Solution: Port Modes : Procedural Mode and Event-based Mode**

The problem above is essentially that application programmers have conflicting expectations. Some prefer a procedural programming style with repeated invocations and polling for network state changes. Others prefer an event-based programming style where program logic is expressed as reactions to events. Fortunately though, programmers tend to choose one or the other. Hence, the idea is to let the programmer choose which semantic is appropriate. In our port abstraction programmers make this decision by specifying the *mode* of a port. There are two available modes, *Procedural mode* and *Event-based mode*. Next we explain how the port mode affects the behaviour of ports.

### 6.6.1 Port Modes for OutPorts

**OutPort: Procedural Mode.** In Procedural mode the invokation of ReadyHandler() is simply omitted if Send() is immediately successful. So, when a successful Send() returns, the OutPort is in the Ready state. This way, repeated invokactions of Send() is supported. If, on the other hand if Send() is not immediately successfull, the OutPort will stay NotReady until ReadyHandler() is executed at some point later. When ReadyHandler() executes, the port is guaranteed to be Ready. Since the ReadyHandler() is never invoked from within Send() the method-nesting problem is avoided.

**OutPort: Event-Based Mode.** The alternative approach is to say that the OutPort should not support repeated Send() operations. Instead, the ReadyHandler() will be invoked exactly once, *after* each Send() operation. In this mode, the return value of a Send() operation is irrellevant. The OutPort will stay NotReady until the ReadyHandler() is invoked, no matter what. This way, the port is always guaranteed to be Ready when the ReadyHandler() executes. Also, there is no nesting problem.



The above figure illustrates the control flow when an application interacts with an OutPort in Procedural mode and in Event-based mode, respectively.

**Procedural mode.** Repeated Send() operations are supported as long as the internal write operation succeed immediately. The return value "ok" indicates this in the illustration. In contrast, if the Send() operation is not able to complete immediately "init" is returned. The black, dotted arrow in the figure illustrates that the Port takes responsibility for completing the Send() operation. When the operation is completed, at some point later, the ReadyHandler() is invoked to signal this event. The bottom half of the figure illustrates that the OutPort supports new Send() operations to be launched from within the the ReadyHandler().

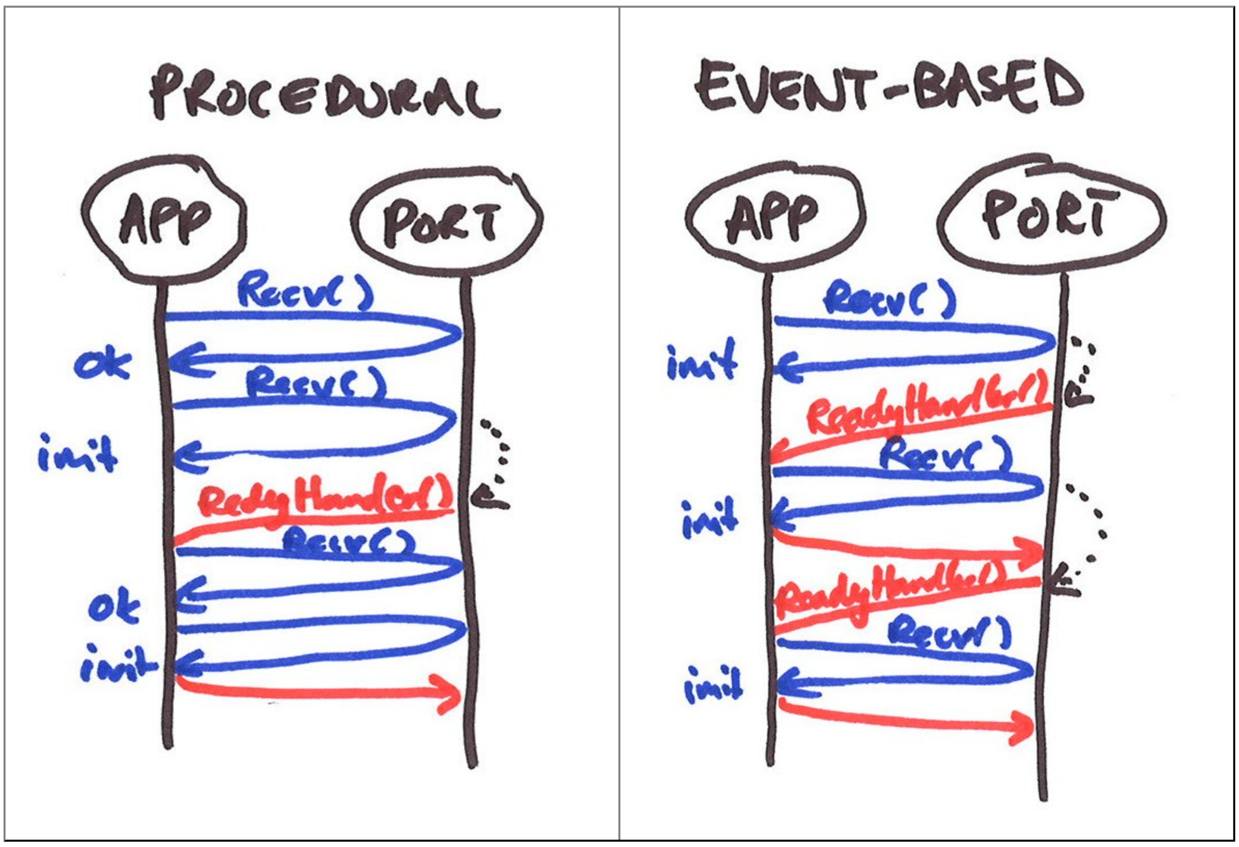
**Event-based mode.** The interaction is initiated by the application by invoking the first Send(). The rest of the application may then be expressed by implementing the ReadyHandler(). Send() return "init" on every invokation. This does not imply that the Send() operation was not immediately completed. Rather, "init" is always returned because the completion of a Send() is implied by the execution of the ReadyHandler(), not the return value of Send(). Attempts to issue repeated Send() operations will be denied by the OutPort.

### 6.6.2 Port Modes for InPorts

The problems that motivated Port modes were presented in the context of OutPorts. However, those issues apply to InPorts as well. We will leave the elaboration of those problems in the context of the InPort, to the reader. Instead we simply describe how port modes affect the behaviour of InPorts.

**InPort: Procedural Mode**. An InPort in Procedural mode supports repeated invokactions of Recv(). The return value of each Recv() operation will indicate whether the InPort is Ready for the next Recv(). If this is the case, the InPort will be Ready as Recv() returns and the ReadyHandler() will be omitted. If, in contrast the InPort is NotReady, it will stay NotReady until the ReadyHandler() is invoked some time later. The InPort is guaranteed to be Ready whenthe ReadyHandler() executes. Infinite method-nesting is avoided.

**InPort: Event-based Mode**. An InPort in Event-based mode does not support repeated Recv() operations. Instead, the ReadyHandler() will be invoked exactly once, *after* each Recv() operation. In this mode, the return value of a Recv() operation is irrellevant. The InPort will stay NotReady until the ReadyHandler() is invoked, no matter what. This way, the port is always guaranteed to be Ready when the ReadyHandler() executes. Also, there is no nesting problem.



The above figure illustrates the control flow when an application interacts with an InPort in Procedural mode and in Event-based mode, respectively.

## 6.7 Port Modes and the Buffer Analogy

Another way to explain the distinction between Procedural and Event-based mode, is to describe it in terms of buffer space. For example, think about an OutPort as being associated with an abstract message buffer, where the buffer size given by the number of message it can hold. Now, by pushing messages into this buffer, the OutPort is playing the role of *producer* in an *producer-consumer* protocol. The role of the *consumer* is played by the underlying network service as it removes messages from the buffer (after having successfully transmitted them on the network).

In general, an application may push new messages into the OutPort continuously, as long as its internal buffer stays non-full. When the buffer is full, the OutPort becomes NotReady, and remains so until there buffer is yet again non-full. The OutPort notifies the application of this condition by invoking its ReadyHandler().

The distinction between Procedural Mode and Event-Based mode can now be understood as an implication of different buffer size. If the buffer size is exactly 1 message, the application may push only one message into the OutPort at once. Then it always has to wait until the ReadyHandler() is exectued. This behavior is equal to what we presented as Event-Based mode. In contrast, if the buffer size is N messages (N>1), up to N messages may be pushed sequentially into the OutPort without causing the OutPort to become NotReady. In particular, the ReadyHandler() will not be executed after every pushed message. Only after the buffer was full and then became non-full will the ReadyHandler() be invoked. This behavior is equal to what we have labelled as Procedural mode.

*Event-based mode corresponds to an abstract buffer of size == 1 message.*
*Procedural mode corresponds to an abstract buffer of size > 1 message.*

In event-based mode an OutPort limits its abstract buffer space by exposing only the 1 message application-level buffer. In procedural mode the OutPort exposes both the size 1 application-level buffer plus the underlying network buffer. Note that an implication of this is that Port mode in some sense depends on message size. If a message is sufficiently large it may fill the abstract buffer space in one go, thereby forcing an OutPort in Procedural mode to behave as if it where in Event-based mode. A similar argument may be provided for the InPort.

The buffer analogy also helps explain what happens to Port modes if the application level buffer is extended. For example, one could easily imagine a Port implementation where the internal buffer size was not 1 message but N (N>1) messages. In fact, the chain simulator used in this project requires such a buffered OutPort.

*Buffered ports are always Procedural mode.*

Note that this does not hold if the semantics of internal events are changed from edgeevents to periodic events. E.g. from BufferJustBecameNonFullEvent to BufferIsStillNonFullEvent.

## 6.8 Message Port API and Summary

Now we can summarise the API of message ports and point out how the programmer benefits from such an abstraction. Both In-Ports and Out-Ports can be instantiated without a network connection. The Port mode is given during initialization.

| **In-Port** | **Out-Port** |
|---|---|
| # Basic API | # Basic API |
| **IsReady**() | **IsReady**() |
| id, message **= Recv**() | **Send**(id, message) |
| **ReadyHandler**(thisPort, next_id) | **ReadyHandler**(thisPort, last_id) |
| # Connection API | # Connection API |
| conn **= SetConnection**(new_conn) | conn **= SetConnection**(new_conn) |
| **HasConnection**() | **HasConnection**() |
| **Reset**() | **Reset**() |
| **DisconnectHandler**(thisPort, error) | **DisconnectHandler**(thisPort, error) |

In summary, this Message Port abstraction hides complexity and facilitates separation of concern. This is especially valuable for applications that multiplex multiple streams of discrete messages across multiple network connections. The chain proxy is one example of such an application, the chain itself would be another. In essence, the Message Port abstraction offers a network programming model where the network is either Ready or Not-Ready to the transfer of discrete messages. The core message-streaming logic of an application may then be specified without reference to anything network related, except the state of the Ports. Furthermore, Message Ports decouple application logic from network-related logic, thereby facilitating separation of concern. For example, the logic that deals with fixing and replacing broke network connections may be isolated from the code that specifies normal application behavior. In particular, this network programming abstraction is helpful both for explaining and implementing the chain proxy.

# 7. Design of the Chain Proxy

With the introduction of the session-start/stop protocols and the Message Port abstraction we may finally address the design of the chain proxy. The content of this section is organised according to the following topics.

First we run through the core logic of the chain proxy. Next we discuss a collection of relevant design topics, both functional and non-functional. The topics include threading, programming models, timeout management, blocking/non-blocking operations, flow control and buffering and notifications. For each topic discussion we show how this relates to our design. Finally, the section is summarised by including a complete listing of the chain API.

## 7.1 The Logic of the Chain Proxy

After having introduced Message Ports, we are ready to present the core logic of the chain proxy. The presentation is structured according to the set of events that trigger activity in the proxy. The chain proxy has four TCP-based message ports, the Head-Out-Port (Updates), the Tail-Out-Port (Queries), the Tail-In-Port (Replies and Notifications) and Head-In-Port (Nothing). In addition it has two UDP-based message ports, Ping-In-Port and Pong-Out-Port. The application delivers messages using the UpdateAsynch(update) and QueryAsynch(query) primitives of the proxy API. The chain proxy delivers Replies to the application by invoking a ReplyHandler() and notifies the application of timeouts using a TimeoutHandler(). The application starts a session by invoking SessionStartAsynch() and terminates the session by invoking SessionStopAsynch(). The proxy invokes SessionStartHandler() and SessionStopHandler() when the chain confirmes that a session started or terminated, respectively.

Here follows the pseudo-code for the chain proxy. The implementation of this logic is structured in a similar way, and is to be found in proxy/chainProxy.py.

```
1. Initialise: The chain proxy is initialised

        Initialise Chain View <HeadIP, TailIP, Length>
                HeadIP and TailIP is not known, Length is 3 by default
        Initialise TailInPort, TailOutPort, HeadOut-Port and HeadInPort
        Initialise PingInPort and PingOutPort.

2. SessionStart: Application invokes SessionStartAsynch(HeadIP)

        Connect to Head --> event New Head-Connection
        Generate RegisterUpdate.
        Invoke UpdateAsynch(RegisterUpdate) --> event Send Update

3. New Head-Connection: New Head-connection established.

        Initialise HeadOutPort and HeadInPort with new Head-connection
                HeadOutPort.SetConnection(conn)
                HeadInPort.SetConnection(conn)

4. SessionStop: Application invokes SessionStopAsynch()

        Generate UnregisterUpdate.
        Invoke UpdateAsynch(UnregisterUpdate) --> event Send Update

5. Send Update: Application or proxy invokes UpdateAsych(update)

        If HeadOutPort is Ready :
                Get new updateID
                Set timeout for expected Reply
                Register Update as Pending
                HeadOutPort.Send(updateID, update)

6. Send Query: Application invokes QueryAsynch(query)

        If TailOutPort is Ready :
                Get new queryID
                Set timeout for expected Reply
                Register Query as Pending
                TailOutPort.Send(queryID, query)

7. TailInPort Ready: A new message is available on the TailInPort

        Get message by invoking TailInPort.Recv()
        If message is Reply --> event Recv Reply
        If message is Notification --> event Recv Notification
```

**8. Recv Reply**: A new Reply is received from InTailPort

```
Cancel timeout for Reply
Detect missing Replies:
If an earlier Reply is missing, force Timeout --> event New Timeout
If Reply is UnregisterReply --> event Recv UnregisterReply
If Reply is UpdateReply invoke UpdateReplyHandler() of Application
If Reply is QueryReply invoke QueryReplyHandler() of Application
If Reply is RegisterReply invoke StartSessionHandler() of Application
```

**9. Recv UnregisterReply**: An UnregisterReply is received from InTailPort.

```
Reset chain view
Reset all Ports
Reset all Timouts
Close Head-connection and Tail-connection
```

**10. New Timeout**: The timeout has expired for some Reply

```
If Reply is QueryReply
    Invoke QueryTimeoutHandler() of Application
If Reply is UpdateReply
    Invoke UpdateTimeoutHandler() of Application
```

**11. Recv Notification**: A new Notification is received from TailInPort

```
Compare Notification <HeadIP, TailIP, Length> with current Chain View.
If HeadIP has changed--> event New Head
    Update Chain View with new HeadIP
If Length has changed
    Adjust calculation of timeouts for Updates
    Update Chain View with new Length
If TailIP has changed
    Update Chain View with new TailIP
```

**12. New Head**: The chain has a new Head

```
Connect to Head --> event New Head-Connection
```

**13. New Tail-Connection**: Tail of the chain connects to the Proxy

```
Cancel Chain Timeout
Replace old Tail-connection
    Invoke TailInPort.SetConnection(conn)
    Invoke TailOutPort.SetConnection(conn)
```

**14. TailOutPort Ready**: TailOutPort completely wrote Query message

```
Invoke QueryReadyHandler() of Application
```

**15. HeadOutPort Ready**: Head-Out-Port completely wrote Update message

Invoke **UpdateReadyHandler**() of Application

**16. PingInPort Ready**: A new Ping is available on the PingInPort

If PongOutPort is Ready: PongOutPort.Send(Ping)

**17. TailInPort Error & TailOutPort Error**: Network error Tail-connection

If Head-connection is down too: Register new Chain Timeout.

**18. HeadOutPort Error & HeadInPort Error**: Network error Head-connection

If Tail-connection is down too: Register new Chain Timeout

**19. Chain Timeout**

```
For each pending Queries and Updates
        --> event New Timeout
Invoke SessionTimeoutHandler() of Application
```

Note that this survey of the chain proxy logic suggests a couple of new methods and upcalls to be added to the chain proxy API.

First, the QueryReadyHandler() and the UpdateReadyHandler() mentioned in event 14 and 15, have not yet been described as a part of the Chain Proxy API. These upcalls signal to the application that QueryAsynch() and UpdateAsynch() are guaranteed to succeed. If the application implements queuing of Queries and Updates, these upcalls may be used to fetch messages from the queues and then send them off directly.

Second, the events 2 and 4 mentions the methods StartSessionAsynch() and StopSessionAsynch(). As the name suggests, these methods allow chain proxies to start and stop a client session.

## 7.2 Threading

As discussed in section 4, a chain client is a program comprised of two parts, the application and the chain proxy. When discussing threading of chain clients we need to take both these parts into consideration. In general, there are two options. The chain client may be a multi-threaded or a single-threaded program.

> **Multi-Threaded**. The application is run by M threads (M >= 1) and the chain proxy is run by N threads (N >= 1).[4]
>
> **Single-Threaded**. The application and the chain proxy together is run by 1 thread.

It would be nice if the chain API supported the development of both multi-threaded and single-threaded chain clients, so we will discuss both these options.

### 7.2.1 Multi-Threaded Chain Clients

An important design objective for the chain API is to avoid placing restrictions on how the application-part of the chain client can be structured and implemented. In this context, this means that M (the number of threads running the application) is considered to be outside our control. Instead, our design need to focus on deciding an appropriate value for N (the number of threads running the chain proxy). However, before we do so, we will briefly discuss an important point regarding the interaction of application threads with proxy thread(s).

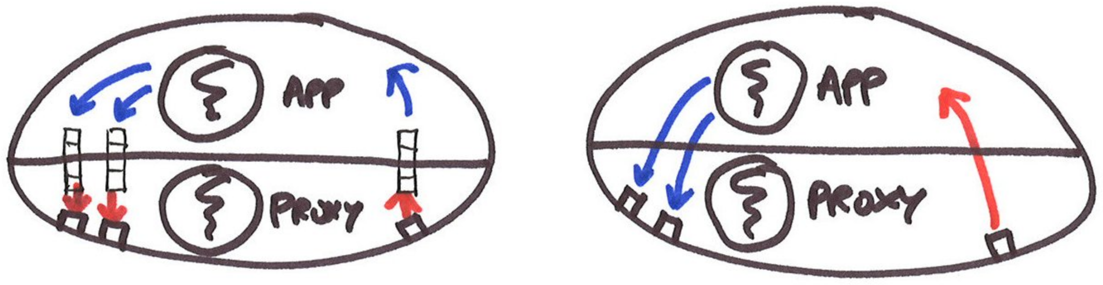
Consider an application that intends to send a request to the chain, a Query or an Update. In a multi-threaded chain client, the application will most likely not run in the same thread as the chain proxy. So, there is a question of how the request should be handed over from the application thread to the proxy thread.

The trivial solution is a traditional queue-based thread-handover. The application thread places the request on a queue. Whenever this queue is non-empty, the proxy thread will pop the request and write it to the network. This is not a good solution, because it requires at least on thread-switch per request. This would hurt the request-reply latency and it would be devastating to the throughput of the streams.

Instead, what we want is to avoid the thread handover all together. That is, we want to allow the application thread to send the request directly. This is possible by allowing the application thread to call right through the proxy, all the way down to the network buffers. We call this *application-write-through*. The Message Ports internal to the chain proxy support this feature. Under normal network conditions this will allow the application thread to fire many requests in sequence, at a high rate. This is exactly the kind of behaviour which is required by pipelining applications.

---

4. In principle, there are other ways to split threads between the application and the chain proxy. For instance, one could imagine a two-threaded chain client where one thread runs one part of the application exclusively, while the other thread runs both the rest of the application and the chain proxy. In this case M would be about 1.5 and N about 0.5. So a slightly more obsessive way to define thread splitting within multi-threaded chain clients would be as follows. (M > 0) & (N > 0) & (N+M >= 2)



The above figure illustrates this idea. In both sketches the proxy thread does work associated with the red arrows. Similarly the application thread does the work associated with the blue arrows. The first solution illustrates a thread handover for all message streams running through the chain proxy.

In the second solution the work is somewhat redistributed between the two threads, so that the thread handover can mostly be avoided. In  fact, the thread-handover is only necessary if the request could not be completed at the first attempt by the application thread. In this case, the proxy thread (associated with the internal Message Port) will take over the responsibility for completing the transfer later (as soon as possible). This will still require a thread-switch, however this may be a rare event under good network conditions.

Avoiding the request handover-buffer, between the application and the chain proxy, also has another positive effect. With the buffer in place sending a request would be an extremely cheap operation, for the application thread. As a consequence, the buffer would fill up quickly and remain close to full most of the time. The time each request would spend in this buffer would add unnecessarily to the latency of the request. In addition, the application would spend much of its time waiting for available buffer space. Instead, with a direct write-through, the pace of the application is naturally adjusted to the realities of the network.

### 7.2.2 Single-Threaded versus Multi-Threaded Chain Proxy

We have chosen a single-threaded design, rather than a multi-threaded design for the chain proxy. Both solutions would probably work. Still, we would like to expose the rationale for choosing one over the other.

A reasonable multi-threaded implementation of the chain proxy would include about three or four threads. For example, one thread associated with each message stream (i.e. each Message Port) and one thread for the management of timeouts. Since these threads would mostly be independent from each other, the overhead of synchronization would be low, and the solution would probably work fine. However, this solution would have some smaller weaknesses.

> **Idle threads are not free**. All these threads would not be busy all the time. Especially, the timeout manager thread would probably sleep most of the time. Furthermore, the threads associated with the Query-stream and the Update-stream would only rarely have any work to do. As discussed above, this is a consequence of application write-through. The appropriate proxy thread is only involved in writing a request to the network, if the application thread failed to do so at the first attempt. Under normal circumstances this might even be rare. So, these threads are mostly idle, yet they continue to absorb resources by being part of the scheduling loop.
>
> **Inflexible Scheduling Policy**. The use of threads is (often unintentionally) a choice of policy with regard to provisioning and scheduling of cpu resources. For example, the chain proxy might be run by 4 threads while the application is run by a single thread. This setup would imply a partitioning of cpu-resources, where the application receives (in the worst case) only about 1/5 of the available cpu-cycles. Now, this is not necessarily a bad idea. After all, many chain clients will be io-bound programs. However, the point is rather that this represents a very specific choice, and a rather inflexible one.

An single-threaded, event-based solution may address these weaknesses. The core idea is that the timeout manager and all the Ports inside the chain proxy are run by the same thread. In addition, by implementing the proxy according to an event-based programming model, it is possible to make the scheduling mechanism much more explicit and available to the application programmer.

The adoption of an event-based programming model requires that the program logic is represented as a set of non-blocking computation-tasks (event-handlers). Each of these tasks is associated with some condition (event). When this condition becomes true (event occurs) this triggers the execution of that task. An event-based program typically includes a scheduling queue where all tasks that await execution are stored. Running such a program will simply involve popping tasks from the queue and executing them serially, often in an infinite event-loop. During task execution new tasks may be created and added to the queue.

This way of structuring a program may come with a slightly higher development cost, but has several desirable properties.

- Preemptive Thread-switching is replaced by cheaper non-preemptive task-switching.
- Blocking operations must be replaced by efficient non-blocking operations.
- There is no overhead associated with thread synchronisation.
- There are no idle threads.
- Provisioning and scheduling of cpu resources is dynamic. A task represents a need for cpu resources. The program therefore assigns cpu resources only where it is actually needed at the time. In contrast, the thread-based solution assigns cpu-resources based on a static, predefined choice implied by the number of threads and possibly their relative priority.

In a multi-threaded chain client, such an explicit cpu-scheduling mechanism can be valuable. The chain proxy will provide a **RunOnce**() method as an entry-point to its internal event-loop. This way, the application programmer may monitor and control in detail the cpu-resources absorbed by the chain proxy. In order to improve responsiveness and reduce latency of the chain proxy, the application may invoke run_once() more frequently. In contrast, if the idea is to prioritise the application logic, run_once() may be called less frequently.

So, in conclusion. We went with a single-threaded, event-based design because it seems to be slightly more effective, while at the same being more flexible when it comes to scheduling of cpu-resources within the chain client.

Naturally, the chain proxy will also provide **Run**() method that iterates the internal event-loop forever. This is handy if the application programmer wants to dedicate one thread to running the chain proxy.

This defines the execution API of the chain proxy.

- **Run**()
- **RunOnce**()

### 7.2.3 Single-threaded Chain Clients

Finally, an additional reason to avoid a multi-threaded chain proxy, and go for an event-based design, is that it keeps the possibility open of running both application and proxy together, in a single thread. If this can be done, this is most likely the most efficient solution.

## 7.3 Programming Models

The chain API should be flexible enough to support a variety of applications. Above we discussed one way of ensuring this; Allowing both multi-threaded and single-threaded applications to integrate well with the proxy. Another important way applications differ from each other is related to the choice of programming model. Application logic may be expressed in an *event-based* manner or in a *procedural* manner.

> **Event-based** programs (as discussed above) typically express their logic as reactions triggered by certain events or conditions. As a consequence, the source code of such programs is often comprised by a collection of handlers, i.e. non-blocking tasks. In addition, if the program logic is sufficiently complex, state machines may become a valuable tool for monitoring program state, and orchestrating important state transition. Event-based programs are typically run from an infinite event-loop.
>
> **Procedural** programs, in contrast, typically express logic as actions rather than reactions. A procedural program may be read mostly as a sequence of statements, thereby making them in general more easy to read. The use of threads is often motivated by a wish to run two logically distinct computations concurrently, while at the same time expressing their logic in a sequential, procedural manner.

Event-based applications and procedural application are not likely to interact with the chain proxy in exactly the same way. For example, consider the challenge of sending a series of Updates to the chain, back-to-back.

The event-based way to structure this logic would be to implement the sending of each Update as a reaction to a specific event. In this case, the appropriate event would be the event that the proxy becomes ready to accept a new Update. If the chain proxy were to expose such an event, the event-based program could simply send the first Update, and have the rest of the Updates being sent as a reaction to the completion of the previous operation. This requirement of event-based applications is part of the motivation for including the following upcalls and setters in the chain API.

- **QueryReadyHandler**()
- **UpdateReadyHandler**()
- **SetQueryReadyHandler**(ReadyHandler)
- **SetUpdateReadyHandler**(ReadyHandler)

UpdateReadyHandler() must be invoked by the chain proxy whenever it *becomes* ready to accept a new Update. Similarly, QueryReadyHandler() is invoked when the proxy *becomes* ready to accept a new Query. Event-based applications may use these handlers, and other handlers, to implement their logic as reactions to events.

The procedural way of structuring this logic is much simpler. The procedural application would simply want to iterate the sequence of Updates and send them one by one. However, this simple logic may be slightly complicated by the fact that the proxy may not always allow Updates to be sent, due to poor network conditions for instance. So, the application would want to check the result of each send operation before the next is initiated. If an Update could not be sent completely, the application will need to take a break and then continue sending Updates at a later point, when the proxy yet again becomes ready to accept new Updates. The UpdateReadyHandler() could be used to trigger this continued iteration of sending Updates.[5]

### 7.3.1 Modes

So, we see that the UpdateReadyHandler() and the QueryReadyHandler() may have utility in both procedural and event-based applications. However, an event-based and a procedural application would not have the same requirements with regard to semantics. The event-based application would rely on the UpdateReadyHandler() to be invoked *every* time an Update was completely sent. In contrast, the procedural application would require the UpdateReadyHandler() to be invoked *only* when the send operation was completed by the proxy, as opposed to being completed by the application on the first attempt.

At this point it should be clear that this issue has been discussed before, in the context of Port modes. Recall that Ports define two modes; Event-based and Procedural mode, to allow the application programmer to choose the suitable semantics. We now realise that the same applies to the chain proxy at large. In order to properly support both event-based and procedural application, the chain proxy will have to define two modes of operation, Event-based and Procedural, and allow the application programmer to decide which one is appropriate.

Fortunately, this is not difficult. In fact, the mode given to the chain proxy at startup need simply be forwarded to its two internal OutPorts. Then the chain API will simply inherit the behaviour of these ports.

### 7.3.2 Timeouts and Tasks

In addition the chain proxy exposes mechanisms that let the application delegate the execution of non-blocking computational tasks (i.e. methods) to the chain proxy. This ability adds four new methods to the chain API.

- **AppendTask**(method, *args, **kwargs)
- toID = **CreateTimeout**(t, method, *args, **kwargs)
- **CancelTimeout**(toID)
- **CompleteTimeout**(toID)

AppendTask() causes the given method to be executed as soon as possible, by the proxy, with the given parameters. CreateTimeout() does exactly same, except it causes the method to be executed not immediately but *t* seconds from now. CancelTimeout() cancels a timeout that was created earlier. CompleteTimeout() forces an existing timeout to expire immediately.

5. If the send operation implemented block-on-send semantics, the programmer need not be concerned with this at all. However, the UpdateReadyHandler() would still be needed to implement such blocking. Block-on-send semantics will be discussed further below.

## 7.4 Timeouts and Lost Replies

First we detail the behaviour we expect from the chain proxy, with respect to lost Replies and timeouts. Next we discuss how the timeout-values internal to the chain proxy may be manipulated by the application programmer.

### 7.4.1 The Rules

The chain proxy keeps record of the state of Replies. A Reply can be *Pending*, *Expected*, *Received*, *Lost* or *Duplicate*. Expected Replies are the next-in-line Reply. Recall, the Query-Reply-Stream is an ordered stream. So is the Update-Reply-Stream.

1. If a Reply arrives before its timeout expires, the Reply is *Received*.
2. If a Reply does not arrive before its timeout expires, the Reply is *Lost*.
3. If a Reply arrives after the timeout has expired, the Reply is *Duplicate*.
4. If a Reply arrives after another Reply with the same ID, the Reply is Duplicate.
5. If a Reply arrives with an ID > Expected_ID.
    - All Replies R, Expected_ID =< R_ID < ID are Lost.

- **Received Replies** are handed over to the application. If the Reply is associated with a non-blocking operation, the ReplyHandler() will be invoked by the chain proxy. If the Reply is associated with a non-blocking operation, the chain proxy gives the Reply to the sleeping thread and wakes it up.
- **Lost Replies** correspond to timeouts. If the Lost Reply is associated with a non-blocking operation, the TimeoutHandler() will be invoked by the chain proxy. If the Lost Reply is associated with a non-blocking operation, the chain proxy wakes up the sleeping thread and indicates that a timeout occurred.
- **Duplicate Replies** are simply dropped.

If more than one Reply is deemed to be Lost in one batch (see point 5), the TimeoutHandlers() will be invoked in the correct order.

### 7.4.2 Managing the Timeout Values

The chain proxy maintains four timeout values. There is a SessionTimeout, an UpdateTimeout and a QueryTimeout. In addition, there is a special RegisterTimeout associated with Register-Replies. The Register-Update is special because it needs to wait for both the head-connection and the tail-connection to be established, in order to complete its journey through the chain.

Application programmers should be able to define values for these timeouts, so that they match the realities of both network conditions and chain load. Furthermore, programmers may want to tune these values, so that the timeouts are neither too short nor too long. The chain API allows the application programmers to do this. Set/Get methods allow all these timeout values to be read and modified at any time.

- **SetQueryTimeout**(timeout)
- **SetUpdateTimeout**(timeout)
- **SetRegisterTimeout**(timeout)
- **SetSessionTimeout**(timeout)
- timeout = **GetQueryTimeout**()
- timeout = **GetUpdateTimeout**()
- timeout = **GetRegisterTimeout**()
- timeout = **GetSessionTimeout**()

In addition, the UpdateTimeout and the RegisterTimeout are sensitive to changes in the length of the chain. Ideally, they should be updated as the chain shrinks or grows. The chain proxy allows this by invoking a ChainLengthHandler(newLength) whenever the chain length has changed. The chain API will define the following setter for this upcall.

- **SetChainLengthHandler**(ChainLengthHandler)

Changes to timeout values do not effect the timeouts of those Queries and Updates that are already sent.

## 7.5 Blocking (Synch) and Non-Blocking (Asynch) Operations

The Chain API primarily offers non-blocking (asynchronous) operations for sending Queries and Updates. This is reasonable, considering that the ability to pipeline requests is so central to the Chain Replication protocol. However, in multithreaded applications, blocking (synchronous) operations may be helpful as well. Especially, if there are dependencies within the stream of requests, the desired request scheduling may for instance be accomplished by mixing blocking and non-blocking operations. For example, if a set of Update requests depend on the result of a preceding Query, that Query may be issued as a blocking operation. Then, as soon as the Query operation returns with the Reply, the set of Updates may be sent as non-blocking operations.

For this reason, the Chain API offers both a non-blocking and a blocking version of the four central methods. Non-blocking methods include the word Asynch in their method names to set them apart from their blocking counter-parts. A complete Chain application may be built using non-blocking operations exclusively, or to the other extreme, blocking operations exclusively.

| | |
|---|---|
| **StartSessionAsynch**(headHost) | **StartSession**(headHost) |
| **QueryAsych**(query) | **Query**(query) |
| **UpdateAsynch**(update) | **Update**(update) |
| **StopSessionAsynch**() | **StopSession**() |

Actually, to present our blocking API correctly, we need to distinguish two types of blocking behaviour; *block-on-send* and *block-for-reply*.

- **Block-on-send** implies blocking (if necessary) until a Query or an Update could be sent. This would be similar to blocking on a socket.send() operation. In the chain proxy all socket operations are non-blocking, still, low level network conditions such as a full send-buffer will stall the Request-stream all the same. In our implementation this condition would translate to an OutPort being NotReady. So, block-on-send semantics will involve blocking the application thread until the appropriate OutPort becomes Ready.
- **Block-on-reply** involves blocking until the Reply is available or until the Request has timed out. This is similar to remote procedure calls [RPC] where an entire Request-Reply protocol is encapsulated within what appears to be a local method invocation.

In our implementation, when an operation is said to be blocking, this means block-on-reply semantics. In addition though, block-on-send semantics are optional. This is especially helpful when one of the two-connections to the chain are disconnected. In this case, an attempt to send a request will not succeed until some time later, when the appropriate connection has been restored.

The chain proxy supports two different block-on-send semantics. Application programmers choose between 1) Limited block-on-send and 2) Unlimited block-on-send. This is accomplished by introducing a *SendTimeout* into the chain Proxy.

- Limited block-on-send: (SendTimeout > 0) Block maximum SendTimeout seconds.
- Unlimited block-on-send: (SendTimeout == None) Block until send is completed.

This introduces two new methods in the chain API.

- **SetSendTimeout**(timeout)
- timeout = **GetSendTimeout**()

SendTimeout  is shared between outgoing Updates and Queries. When the SendTimeout expires for a given request, (INIT, requestID) is returned. This tells the programmer that the transfer of the request was initiated, but could not be completed at this point. The chain proxy takes responsibility for completing the message transfer as soon as the network connection allows it.

Note also that the SendTimeout does not overlap with the QueryTimeout or the UpdateTimeout. This is because the QueryTimeout and the UpdateTimeout is started *after* the message has been completely sent, i.e. after the SendTimeout has been cancelled.

## 7.6 Flow Control and Buffering

The Query-stream and the Update stream run *out* of the chain proxy. The Reply-stream runs in the opposite direction. Our discussion of flow control and buffering is split between outgoing and incoming streams.

### 7.6.1 Query Stream and Update Stream

There will be problems in the chain (e.g. failed nodes) and as a consequence the message throughput of the chain will vary. These fluctuations are visible to the chain client only in the effect that they have on the performance of the head-connection and the tail-connection. For example, if the throughput of the chain is completely halted at some point, this will eventually cause the Head of the chain to stop accepting new requests. This again causes the send-buffers of the chain proxy to fill up, provided of course that the proxy keep sending new Update requests.

The chain proxy could do two things in response to this situation. One strategy would be to try to hide this condition from the chain application. By buffering requests the chain proxy could pretend there was no problem, and hope that the problem would go away shortly. If it did not go away, the knowledge of this problem would not be propagated to application until the proxy buffers were filled too.

The other strategy would be to not buffer anything, thereby letting the application know immediately that the chain is not accepting new requests. This is the approach we have chosen for the chain proxy. There are several reasons for this.

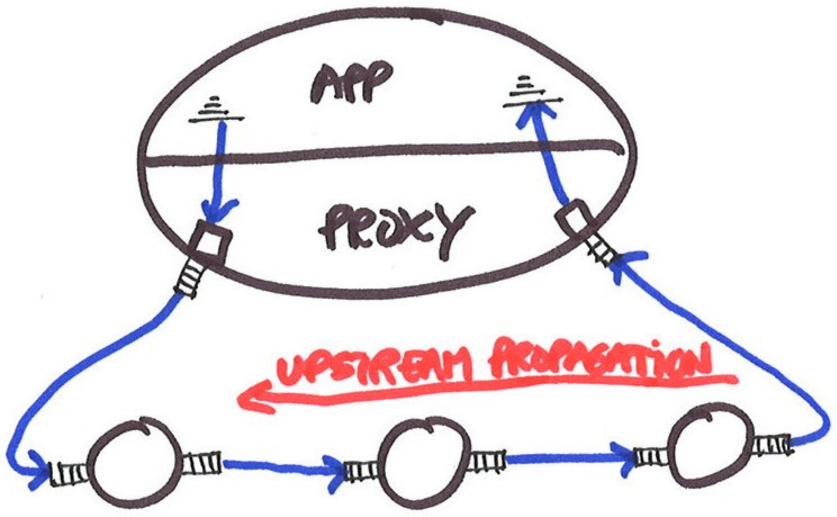
- Buffering in the chain proxy adds unnecessary latency to all request.
- Reduced throughput in the chain may have several causes. For instance, it could be a consequence of high load induced by a large number of simultaneously chain clients. These are not necessarily transient conditions that can be easily masked.
- The chain application is the source of the request load. When there is a problem in the chain, it makes sense to propagate knowledge of this upstream, all the way to the source of messages, the chain application.
- The application may want to define application-specific reaction to such situations.
- The cost and the result of the send-operations should be visible to the application.
- Buffering may also complicate reactions to timeouts. If the application needs to do some kind of roll-back as part of recovery after a timeout, it is helpful to know exactly which messages where actually sent to the chain. A possibly full request buffer, hidden internally by the chain proxy, may complicate roll-back logic. Incidentally, timeouts are not less likely when the chain stalls.
- Application programmers that want a send-buffer may still implement this on top of the chain API.

### 7.6.2 Reply-Stream

So far we have looked at reduced throughput in chain, and the effects that this has on the chain proxy. However, we also need to consider the reverse situation. That is, the chain client is not able or willing to receive Replies at the same rate that the Tail would like to output them.

This would not be an unlikely situation if the chain API was to adopt the InPort API. The chain application would then actively invoke RecvReply() whenever it wants to receive a new Reply. As part of that operation, the next Reply would be read off the tail-connection, if available. Now, this kind of Reply-delivery mechanism would have some problematic consequences.

First, in a sense it is now up to the application to decide the throughput of the chain. If an application does not invoke RecvReply() at an appropriate rate, upstream buffers will start to fill up. It is even possible that this condition will propagate all the way through the chain, until it reached the source of the stream, the application. If so, the outgoing Query-stream and Update-stream would eventually stall. This will depend very much on the details of the flow control mechanism implemented in the chain. However, if indeed this is the case, it is a serious situation. The application has succeeded in reducing the throughput of the entire chain. This is of course unacceptable, unless the chain only has one client. The figure below attempts to illustrate the idea that buffer overflow is a condition that may propagate upstream in the chain.



Second, and perhaps more likely, this condition is not allowed to propagate upstream in the chain. Instead the Tail of the chain will start dropping Replies as soon as the buffer associated with a given client fills up. The result is that timeouts will start to go off in the chain proxy. This in some sense solves the problem then and there. However it will make life harder for the application programmer. Given a sudden burst of timeouts, it is not clear that the correct cause of that will be easy to spot. Also, timeouts may trigger new waves of retransmitted requests, thereby adding to the problem. The fundamental problem here is that the application programmer lacks an effective mechanism for balancing the rate of the Query-stream, the Update-stream and the Reply-stream.

Finally, with this receiver-driven Reply-mechanism, it would be impossible for Notifications and Replies to share the tail-connection. This is because Notifications are protocol messages rather than application messages. The chain proxy depends on Notifications arriving quickly, especially in the event of a Head failure. Notifications should not be stuck in a buffer just because the chain application does not receive Replies at an appropriate rate.

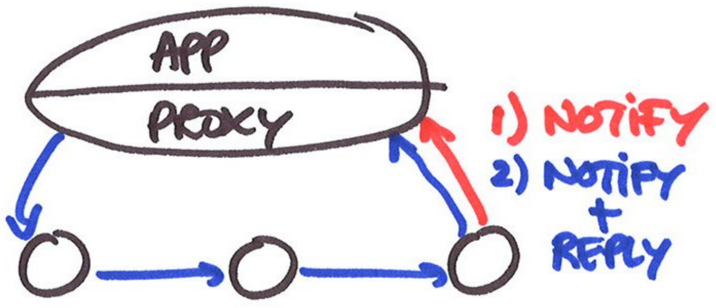
So, we want to avoid this solution. Instead we suggest a best-effort solution where the chain proxy reads Replies aggressively off the tail-connection and hands them directly to the application by invoking ReplyHandler(Reply). The chain proxy will never consider whether this is appropriate or not. In addition, it is part of the API specification that this is supposed to be a non-blocking method. We argue that this solution is better, because it forces the application programmer to implement the buffering. This way, if Replies must be dropped due to limited buffer space in the application, combined with high Reply rates, at least the application programmer can see what happens and take appropriate action.

## 7.7 Notifications and Replies on a Single Connection

This far, we have taken it for granted that Replies and Notifications share a single connection, the Tail-connection. Though, in the above discussion regarding flow control and buffering it became clear this design feature is not without its problems.

One problem is that Notifications can be delayed because the Replies are queued up at the Tail of the chain. One solution would be for the Tail of the chain to allow Notification to sneak in line. Chain Replication keeps messages ordered in stream, but logically Notifications are not really part of the Reply-stream. Recall that the Notifications used to be delivered by the master quite independent of the Reply stream. It was only for convenience that we earlier suggested to merge the Reply stream and the Notification stream.

Now, allowing a Notification to go in front of the application-level buffer may help, but there are still problems. The send-buffer of the tail-connection may be full, so the Notification will not be able to sneak past those Replies that have already been committed to the connection. Big messages are a related problem. If the Tail is busy transfering a huge Reply to the client, Notifications will have to wait till this transfer is completed.



As the figure suggests, the solution to these problems is to set up another tail-connection, dedicated to transmitting Notifications. This indeed solves the problem for the client, but the chain the solution has some disadvantages too. First, by introducing a new connection the entire architecture of the system becomes slightly more complicated and slightly less elegant. Second, this may create a problem of scalability. The Tail of the chain already needs to maintain one connection for each client in the system. Adding a new tail-connection would double the number of connections.[6] This would limit the scalability of the system, with regard to the number of clients. In addition it would add to the overhead whenever all these connections need to be reestablished after a failure.

Ultimately, this will have to be a design choice made by the chain-designer. For this reason, our design of the chain proxy allows for both solutions. The chain proxy is set up to accept two tail connections, on port 5556 and 5557. If the Tail connects to the latter port, the connection is assumed to be used exclusively for Notifications. However, the chain proxy will still accept Notifications on the original tail-connection. Also, the chain proxy does in no way depend on the second tail-connection, so the Tail may shut it down if it is not needed.

6. In fact, a chain of length one would be required to maintain 3 connections per client.

## 7.8 The Complete Chain Proxy API

Finally, we are able to sum up and comment the complete chain API.

**Non-blocking (Asynchronous) API**

- (status, registerID) = **StartSessionAsynch**(headHost)
- (status, queryID) = **QueryAsych**(query)
- (status, updateID) = **UpdateAsynch**(update)
- (status, unregisterID) = **StopSessionAsynch**()

**Blocking (Synchronous) API**

- (status, registerReply) = **StartSession**(headHost)
- (status, queryReply) = **Query**(query)
- (status, updateReply) = **Update**(update)
- (status, unregisterReply) = **StopSession**()

**Upcall Setters**

- **SetQueryReadyHandler**(ReadyHandler)
- **SetUpdateReadyHandler**(ReadyHandler)
- **SetQueryTimeoutHandler**(TimeoutHandler)
- **SetUpdateTimeoutHandler**(TimeoutHandler)
- **SetQueryReplyHandler**(ReplyHandler)
- **SetUpdateReplyHandler**(ReplyHandler)
- **SetSessionStartHandler**(SessionHandler)
- **SetSessionStopHandler**(SessionHandler)
- **SetSessionTimeoutHandler**(SessionHandler)
- **SetChainLengthHandler**(ChainLengthHandler)

**Upcall Handler Types**

- **ReadyHandler**()
- **TimeoutHandler**(requestID)
- **ReplyHandler**(requestID, reply)
- **SessionHandler**()
- **ChainLengthHandler**(chainLength)

**Timeout API**

- **SetQueryTimeout**(timeout)
- **SetUpdateTimeout**(timeout)
- **SetRegisterTimeout**(timeout)
- **SetSessionTimeout**(timeout)
- **SetSendTimeout**(timeout)
- timeout = **GetQueryTimeout**()
- timeout = **GetUpdateTimeout**()
- timeout = **GetRegisterTimeout**()
- timeout = **GetSessionTimeout**()
- timeout = **GetSendTimeout**()

**Status API**

- (starting, live, stopping) = **SessionStatus**()
- (head, tail) = **ConnectionStatus**()
- length = **ChainLength**()
- bool = **IsQueryReady**()
- bool = **IsUpdateReady**()

**TaskScheduler API**

- **AppendTask**(method, *args, **kwargs)
- toID = **CreateTimeout**(t, method, *args, **kwargs)
- **CancelTimeout**(toID)
- **CompleteTimeout**(toID)

**Execution API**

- **Run**()
- **RunOnce**()

### 7.8.1 Initialization

The chain proxy is initialized with the appropriate mode given. The choice of mode affects the behaviour of all four methods in the asynchronous API, along with the two upcalls that accompany them; QueryReadyHandler() and UpdateReadyHandler().

- **EVENT_BASED** (1) : The chain proxy will  with event-based mode.
- **PROCEDURAL** (2) : The chain proxy will run in procedural mode.

### 7.8.2 Return Values

Common to both the synchronous and the asynchronous API is that all methods return two values, *state* and *result*. The result value is the value you would excpect from a successful invokation of the method. The state value may contain information about how an operation was performed, or what went wrong. There are four different state values.

**NOOP** (0): The attempted operation was not performed. For instance, this would be the case if the application attempts to send a Query while the proxy is not ready to accept a new Query.

**INIT** (1): The attempted operation was initiated, but did not complete directly. This would be the case if a Query could not be written completely by the application, at the first attempt. This means that the chain proxy will take responsibility for completing this Query later. It also indicates that the chain proxy will not be able to accept more Queries from the application, until after the QueryReadyHandler() has been invoked.

**OK** (2): The attempted operation was completed directly. This indicates that the chain proxy is ready to accept the next request right away.

**TIMEOUT** (3): The attempted operation timed out.

The synchronous and the asyncronous API use these values a bit differently though. The returned values may also depend on whether the chain proxy runs in Event-Based or Procedural mode.

**The asynchronous API**

- (NOOP, None) is returned: The operation could not be performed.
- (INIT, RequestID) is returned:
  - Procedural mode: The operation was initiated, but could not be immediately completed. The chain proxy is not ready for the next request (of the same type).
  - Event-based mode: The operation was initiated. In addition, it may well have been completed too. The chain proxy is not ready for the next request (of the same type).
- (OK, RequestID) is returned:
  - Procedural mode: The operation was immediately completed. The chain proxy is ready for the next request (of the same type).
  - Event-based mode: OK is never returned, always INIT.

**The synchronous API**

- (NOOP, None) is returned: The operation could not be performed.
- (INIT, RequestID) is returned: The operation timed out (SendTimeout) before being able to completely sending a request.
- (TIMEOUT, None) is returned: The operation timed out (QueryTimeout or UpdateTimeout).
- (OK, Reply) is returned: The operation succeded and the Reply is available.

Neither the ReplyHandler() nor the TimeoutHandler() will be invoked for synchronous operations.

### 7.8.3 Handler Upcalls

- All handlers may be implemented by the application.
- All handlers must be non-blocking and should be cheap operations.
- All handlers are invoked in the thread context of the thread running the core loop of the chain proxy.

# 8. Implementing the Message Port Abstraction

The chain proxy is implemented on top of the Message Port abstraction. Hence, before we discuss the implementation of the chain proxy it makes sense to have a look at the implementation of Message Ports.

The Message Port abstraction is especially helpful when multiplexing streams of discrete messages over multiple connections. A Port can be thought of as a thin wrapper around a network endpoint, i.e. a socket. Port activity is triggered from above by applications invoking Send() and Recv() operations, and from beneith by a select-based socket-monitoring loop. In this sense Ports are similar to *dispatchers* in the Python *asyncore* module. However, Ports are a more high-level abstraction. Most notably, Ports transfer complete messages rather than bytes in a byte stream.

## 8.1 A Single-Threaded, Event-based Execution Model

In accordance with the design discussion in section 7, we have chosen a single-threaded, event-based execution model for the message port implementation.

Recall that Ports accept processing responsibility on behalf of the application. For instance, if the application can not write an entire message to the network, at a given time, the OutPort takes on responsibility for writing it later. To enable Ports to do such processing, decoupled from the control flow of the application, the trivial multi-threaded solution would be to associate a thread with each Port.

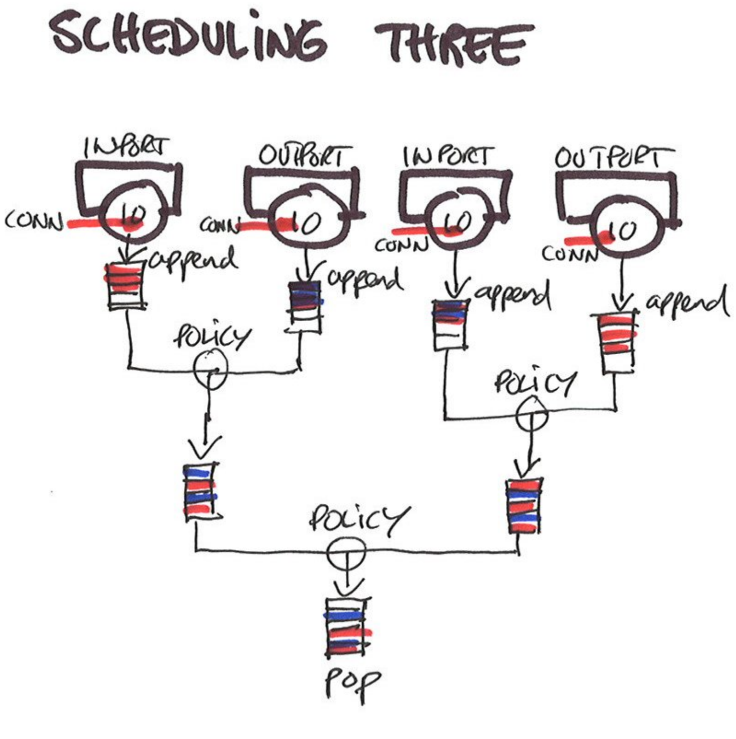
Instead, we want an event-based design where all the Ports to share a single thread. We call this single thread the TaskScheduler. Ports then represent their processing responsibilities as non-blocking Tasks, and forward such Tasks to the TaskScheduler. The TaskScheduler essentially does what its name suggests, it pops Tasks from a queue and executes them serially, in an infinite loop. In addition, it monitors all the network connections using *select*(). Whenever a connection is ready for writing or reading, a new Task is generated. The Port associated with the connection defines the appropriate logic for that Task.

The motivation for this single-threaded, event-based design should be clear from the previous discussion of the same topic, in the context of the chain proxy. However, the design of the Port abstraction is not only motivated by its use in the chain proxy. Rather, Message Ports are developed with a wider class of applications in mind. These are applications that multiplex numerous streams of messages over numerous network connections, simultaneously. The nodes of the chain protocol would be a prominent example of such an application.

This class of applications additionally requires the Port abstraction to be scalable, in the number of Ports and connections. Fortunately, the case for a single-threaded, event-based design is only strengthened by this requirement. A multi-threaded design would typically associate a thread with every Port, or every connection. This would be problematic as the number of Ports and connections grow high, and the overhead of thread scheduling becomes more significant. In contrast, with a single-threaded solution the number of Ports and connections would be slightly irellevant. Instead, the load on the application would be defined only by the combined throughput of all the message streams, as it should be.

The single-threaded, event-based design of the Ports abstraction also yield some interesting capabilities when it comes to scheduling policies. In our implementation the default cpu-scheduling policy of the Port abstraction is open to modification by the application programmer. The default behaviour of the TaskScheduler is to pop Tasks from a single Task queue, in the order they were appended. However, if the programmer wants to prioritze certain Ports over others, say in the event of congestion, this is possible. Prioritizing Ports translates to prioritzing Tasks within the Task Scheduler (in the event that there are multiple Task in the queue at the same time). This need not be difficult. The TaskScheduler and all the Ports are associated with a single Task queue each. If they all reference the same Task queue, this amounts to the default solution; All Ports append Tasks to the same Task queue, and the Task Scheduler pops them off and executes them. However, to improve on the default solution, the application programmer may define a hierarchical tree structure that transports Tasks from the leaf nodes and down to the root node. The TaskScheduler will then pop Tasks off the root, while each Port append Tasks to a single leafnode (given at setup). Branching points in this tree structure may then be used to implement priority scheduling policies. This way application specific scheduling policies may be implemented. For example, OutPorts may be prioritised over InPorts, or messages streams carrying control messages may be prioritised over message streams carrying data. The requirements to such a data structure would be as follows.

- Leaf nodes implement *appendTask*(task).
- Root node implements *popTask*().
- Tasks appended to a specific leaf will be popped at the root in the same order.
- Popping at the root will always return a Task, unless there are no Tasks in the tree.
- Implementation needs to be thread-safe.

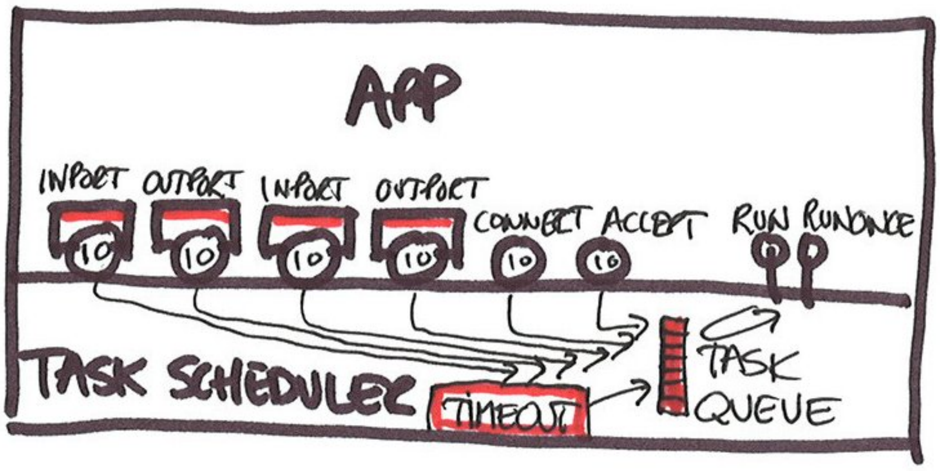


This is similar to the concept of Event Scheduling Trees (EST) in Vortex [VTX]. The chain proxy does not implement such customized scheduling. However, in the prototype implementation of the chain replication protocol we have found this feature to be valuable.

## 8.2 Task Scheduler and IoObjects

The introduction presents Ports as tightly coupled to the TaskScheduler. In fact, the TaskScheduler supports more than just Ports alone. More generally, the TaskScheduler supports IoObjects. A Port is merely a wrapper around an IoObject. Before we discuss the implementation of Ports, we need to present the TaskScheduler and IoObjects in more detail.

An IoObject is essentially an object that is associated with a filedescriptor (fd) and a Task queue (TQ). Using the filedescriptor, IoObjects can be monitored for input or output events using select(). The TaskScheduler does this by polling select() regularly. Then, as a filedescriptor becomes readable/writeable, the TaskScheduler creates a read/write-Task and appends it to the TaskQueue of the appropriate IoObject. Each IoObject implements its own read/write operations in methods io_READ() and io_WRITE(). The read/write Task created by the TaskScheduler simply references the ioObject instance and the appropriate method.



This figure illustrates an application that interacts with IoObjects (circles) and with Ports (squares on top of circles). All IoObjects and Ports are associated with the global TaskQueue, thus they append Tasks to this queue when they need the TaskScheduler to do some work on their behalf. The core loop of the TaskScheduler simply pops Tasks off this queue and execute them. The loop has two phases.

- First, it polls for io-events on behalf of IoObjects. A ReadSet and a WriteSet keep track of which IoObjects need what kind of monitoring. IoObjects may chose to enter or leave the ReadSet or the WriteSet at any time. For instance, this allows an InPort (IoObject) to stay in the ReadSet only when messages are expected to arrive. Polling of multiple IoObjects may cause multiple Read/Write Tasks to be appended to the Task queue.
- Next, the TaskScheduler pops Tasks from the Task queue, one by one, and executes them sequentially. The execution of a Tasks may spawn new Tasks. The TaskScheduler continues until the Task queue is emptied.

There are to ways to run the TaskScheduler. Invoking its run() method will cause it to loop forever, until some other thread invokes shutdown(), or until some exception is raised (e.g. KeyboardInterrupt). This method is well suited for a multi-threaded application, where the TaskScheduler is running in a thread of its own. The other way is to invoke run_once(). This causes the TaskScheduler to run only a single pass through the core loop described above. This gives further control over program-wide scheduling to the application programmer. For example, if the application too is implemented in an event-based, non-blocking fashion, it is possible to run both application and TaskScheduler in a single thread. This would simply require interleaving the execution of application logic with the TaskScheduler. The responsiveness of the TaskScheduler depends on run_once() being executed frequently.

The implementation of the TaskScheduler and the IoObject is found in *ports/taskScheduler.py*

## 8.3 TCP-based IoObjects

Our TaskScheduler implementation currently includes four types of IoObjects associated with TCP communication. UDP-based IoObjects are also available, but here we limit the discussion to those based on TCP.

> **TcpAcceptObject**. The TcpAcceptObject wraps a tcp socket, set to listen on a given port. When a client connects, the socket becomes readable. The io_READ() method does the non-blocking call to accept(). The new socket is handed to the program by invoking the io_ACCEPTED(sock) method. The application programmer needs to implement this method. TcpAcceptObject is always in the ReadSet.
>
> **TcpConnectObject**. The TcpConnectObject wraps a tcp socket, for which a non-blocking connect() has been performed. When the socket becomes writeable, the socket is successfully connected. io_READ() simply hands the new socket over to the program by invoking io_CONNECTED(sock). The application programmer needs to implement this method. TcpConnectObject is in the WriteSet until the socket is connected. After this point, the TcpConnectObject has no function, so it just unregisters from the TaskScheduler. At this point, the socket typically is passed to a TcpWriteObject and/or a TcpReadObject.
>
> **TcpWriteObject**. The TcpWriteObject wraps a connected, non-blocking tcp socket. The io_WRITE() method implements a protocol for writing a single message (header and payload) to the socket. This may succeed in a single invokation. However, if the message was only partially written, the next invocation will continue where the previous left, until at some point the message is completed. The TcpWriteObject stays in the WriteSet only when a partially written message needs to be completed.
>
> **TcpReadObject**. The TcpReadObject wraps a connected, non-blocking tcp socket. The io_READ() method implements a protocol for reading a single message (header and payload) off the socket. This may succed in a single invocation. However, if the message was only partially read, the next invocation will continue where the previous left, until at some point the message is completed. If zero bytes could be read, this too counts as a partial read. The TcpReadObject enters into the ReadSet only after io_READ() returns without completing a message. So, if a message is expected to arrive some time in the future, invoke io_READ(). If the message has already arrived io_READ() will be successful. Or, if it has not, io_READ() will read nothing, but enter the ReadSet as a consequence. Then later, when the message does arrive, the TaskScheduler will invoke the next io_READ().

The message protocol implemented by TcpWriteObjects and TcpReadObjects requires an identifier (integer) to be supplied with every message. For example, io_WRITE(id, message) will start the transfer. The purpose of the id is to allow logging of internal events to be associated with specific messages. The message protocol specifies that each message has an 8 byte header (length, id) followed by the payload. Length is the combined length of header and payload.

TcpWriteObjects and TcpReadObjects expose internal events by executing upcalls with relevant information. TcpWriteObjects invoke WriteDoneHandler() when a message has been completely written. Similarly, TcpReadObjects invoke ReadDoneHandler() when a message has been completely read. Both TcpWriteObject and TcpReadObject expose error-events by invoking CloseHandler(). This means that an error is detected while attempting to read or write the socket.

TcpWriteObjects and TcpReadObjects are not initiated with a socket. Instead sockets can be inserted and removed safely from the IoObject, using InsertSock(sock) and RemoveSock() methods. It is even safe to start writing a message before a socket is provided. TcpWriteObject will start writing the message as soon as a socket is provided.

The implementation of all tcp-based IoObjects is found in *ports/tcpObjects.py*

## 8.4 Timeouts

The Task-based execution model of the TaskScheduler is not only useful for handling multiple network connections. In particular, we have used it to provide a Timeout service within the TaskScheduler. In order to have a specific Task executed, at a specific time, the programmer may simply register this desire with the TaskScheduler. The TaskScheduler then takes responsibility for executing the task, as the timeout expires. The implementation of this is not difficult, though it requires a new phase to be added to the core loop of the Task Scheduler. In addition to polling for io-events the TaskScheduler must now also poll for timeout-events. As a timeout is detected, the Timeout-Task is appended to the appropriate Task queue, like any other Task. All Timeouts are kept in a sorted heap. This makes polling for timeouts a cheap operation. Registered Timeouts may be cancelled or completed before they are due.

Support for Timeouts proved valuable when implementing both the chain proxy and the chain simulator. In the chain proxy, all logic related to timeouts of Queries and Updates is implemented using this service. The chain simulator uses timeouts to simulate added latency for Update requests.

*Implementation of the timeout service is found in ports/taskScheduler.py*

## 8.5 Message Ports

Now, with some intimate knowledge of the TaskScheduler and the IoObjects, the presentation of the Port implementation will be much simplified. An OutPort is just a thin wrapper around a TcpWriteObject. Similarly, an InPort is a thin wrapper around a TcpReadObject. What these wrappers do is to implement the Port interface and to manage the state (Ready/NotReady) of the Port. The state of a Port corresponds mostly to the protocol state (i.e. MessageInProgress/MessageNotInProgress) of the underlying IoObject. However, the corresponce is not perfect, which will be clear when we step through the process of sending a message using an OutPort.

```
def Send(self, id, payload):
        if self.READY:
                self.READY = False
                res = self.io_WRITE(id, payload)
                if res == 1 and self.MODE == PROCEDURAL :
                        return 1
                else : return 0
        else : return -1
```

The application invokes Send() with a new message to be sent. If the Port is not Ready at this point, the request will be ignored and the method will return -1 to indicate this. If, however, the Port was Ready a new message transfer is initiated. The Port indicates this by immediately becoming NOT_READY. At this point an attempt to write the message is made using the underlying TcpWriteObject. This shows that the OutPort supports the concept of application write-through, discussed earlier.

A return value of 1 serves as an indication that the Port is immediately Ready to accept a new Send() operation. Two conditions must be met for this to occur. First, the message must be completely written, at the first attempt. Second, the Port must be in Procedural mode. Recall, Procedural mode means that the Port may support repeated Send() invocations, but that the ReadyHandler() will not be executed if the write operation was immediately successful. The state transition, from NOT_READY back to READY is not evident in the above code sample. This is because it occurs within the io_WRITE() method, as will be explained next.

```
def HandleWriteDone(self, caller, id, payload):
        if self.MODE == PROCEDURAL and caller == "Send":
                self.READY = True
                # Do not invoke ReadyHandler
        else:
                # Schedule Delayed Invocation of ReadyHandler
                self.READY_DELAYED = True
                self.TaskQueue.appendTask(InternalReadyHandler)
```

HandleWriteDone() is an upcall from the TcpWriteObject within the OutPort. It is invoked from within the io_WRITE() method, and it signals that a message has been completely written to the network connection. So, if the message was completely written at the first attempt, the execution of this HandleWriteDone() will be nested within the execution of the Send() method. Alternatively, the write operation will be completed as part of the Task execution of the TaskScheduler, in which case the HandleWriteDone() will be invoked by the TaskScheduler. It is important to be able to recognize the caller of this method, for two reasons.

First, an invocation of HandleWriteDone() within Send() raises the issue of infinite nesting. As discussed earlier, the ReadyHandler() is implemented by the application programmer and may well invoke new Send() operations. Potentially this could cause infinite method nesting leading eventually to stack overflow. Instead, we want to have the ReadyHandler() execute logically *after* the Send() has completed. Fortunately, this is not difficult, with the TaskScheduler at hand. Instead of invoking the ReadyHandler() directly within HandleWriteDone(), the ReadyHandler() is wrapped in a Task and appended to the Task queue of the underlying TcpWriteObject. The TaskScheduler now becomes responsible for the delayed execution of the ReadyHandler()[7] .

Second, if the OutPort is in Procedural mode and the message is completely written, the Port re-enter the Ready state immediately. Still, the ReadyHandler() is not supposed to be invoked. Within the HandleWriteDone() method this condition is true only when HandleWriteDone() is called from within the Send() method.

```
def InternalReadyHandler(self):
        self.READY_DELAYED = False
        self.READY = True
        # Execute ReadyHandler
        self.ReadyHandler(self)
```

InternalReadyHandler() implements the state transition from NOT_READY to READY just before the "real" ReadyHandler() (implemented by the application programmer) is invoked. This way, we ensure that the OutPort is always READY when the ReadyHandler() executes.

The ReadyDelayed flag is added to the implementation to reflect that internally, Ports in fact have three possible states, not two, as presented so far. The three states are:

| | | |
|---|---|---|
| 1) | **NOT_READY** | State=NOT_READY, ReadyDelayed = False |
| 2) | **SOON_READY** | State=NOT_READY, ReadyDelayed = True |
| 3) | **READY** | State=READY, ReadyDelayed = False |

State 2) is the new one and it means that the message is completely written, but that the transition from NOT_READY to READY is delayed. In essence, the InternalReadyHandler() is sitting in the Task queue of the TaskScheduler, awaiting

7. Unfortunately, this can be problematic in a multi-threaded program. For instance, if a thread switch occurs within the OutPort.Send() method, after the ReadyHandler has been appended to the Task queue, but before the return statement of Send() has executed, there is a real risk that the ReadyHandler() will execute before the Send() operation has completed. There seems to be no good way around this problem. The best we could do was to delay the queuing of the ReadyHandler-Task until the very end of the Send() method. However, this still means that the application programmer must be aware of this issue, and must take the necessary precautions.

execution. This new state is not visible to the programmer. It is however vital for preserving the internal integrity of Ports during the SetConnection(sock) operation.

The InPort has an analogous structure to it. This is left as an exercise to the reader. The implementation of InPort and OutPort is found in *ports/tcpPorts.py*. The InPort is implemented by the InTcpPort class while the OutPort is implemented by the OutTcpPort class.

## 8.6 Message Port Upcalls

In addition to the ReadyHandler, the Ports define several other Handlers in order to expose internal events. See the Port API for details. Most of these Handlers reflect internal events in the underlying IoObject, but some Handlers reflect internal events of the Port. For exampel, the OutPort has a SendHandler() to indicate that the Port accepted a new send operation. Similarly, the InPort has a RecvHandler(). The NewConnectionHandler() is shared by both InPorts and OutPorts, and it signales that SetConnection() was invoked on the Port, with a new socket connection. Common to all these Handlers is that they are executed by the TaskScheduler, not by the application. The motivation for this is simplicity. Some of these Handlers, in some cases at least, *must* be executed by the TaskScheduler, and *no* Handler *must* be executed by the application. So, it reduces the complexity of the interface if we just require all Handlers to be executed by the TaskScheduler. Again, implementing this is easily done by wrapping a Handler in a Task and appending it to the Task queue.

## 8.7 Failing TCP Connections

Applications that benefit from the Port abstraction are typically multiplexing multiple network connections simultaneously. The ability to detect failures and remove broken connections may at least be very practical, if not strictly necessary. For this reason, the Port abstraction implements best-effort failure detection. When a connection failure is detected by the TaskScheduler or by some IoObject, the DisconnectHandler() will be invoked on all Ports (i.e. IoObjects) associated with the failed connection. This allows the application programmer to specify application specific measures to the event of a network failure. Failures are reported when:

- A connection is explicitly closed by the peer or by the Port (IoObject) itself.
- A connection fails during a read or write operation.
- A read or write operation is attempted on a failed connection.

Note that if a given connection fails due to peer failure, or peer closing the connection, the failure may go undetected until the next read or write operation. Also, if the connection is not in the ReadSet of the TaskScheduler the failure will go unnoticed. If this best-effort mechanism for failure detection is not satisfactory, the application programmer must take the appropriate measures. For example, a ping-pong protocol may well be built using the PortAbstraction. Since UDP may be preferable for this purpose, compared to TCP, the PortAbstraction implements UDP-based Ports as well.

A reasonable application-level respons to the failure of a connection is to try to set up a new one. The Port implementation allows failed connections to be safely replaced by a new connection, using SetConnection(conn). This ability is discussed in section 6. By safe replacement we mean that the integrity of the the Ready/NotReady protocol is preserved. To do this, the implementation of SetConnection() must be careful not to interfere with the state transitions, while at the same time restarting reading and writing activites within the Port. From the perspective of the application programmer, it is crucial that the Port is able to restart or continue the processing resposibilites it has previously accepted. The implementation of SetConnection is able to do this. Sensitivity with respect to the internal state SOON_READY (discussed above) is required to get this right.

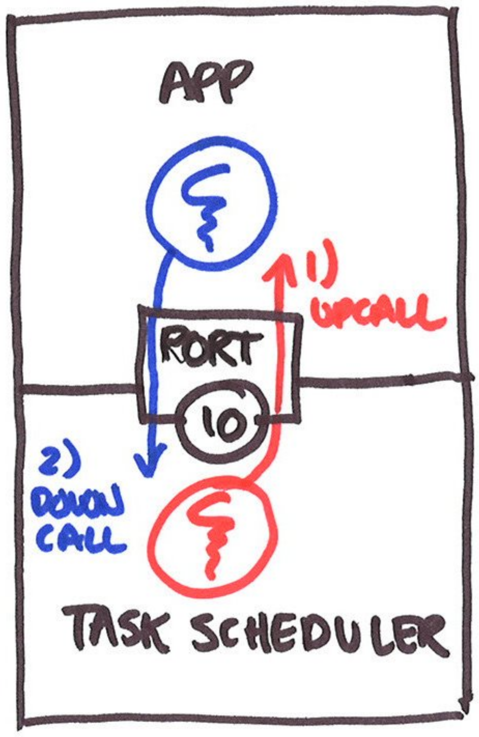
The chain proxy makes use of error detection on tcp-connection in order to clean up failed connections. Still, neither the connection-abstraction associated with tcp, nor the fail-detection ability is essential when implementing chain clients. Especially, the liveness of a client-session does not depend on the liveness of any TCP connection. Furthermore, chain clients do not do failure detection themselves. Instead they rely on Notifications from the chain whenever a failure has been detected and resolved. The Notification may demand a new connection to be created. Only after this connection is successfully established is it given to the approprite Port, using SetConnection().

## 8.8 Thread-Safety of Ports and the TaskScheduler

Multi-threaded programs require the Port implementation to be thread-safe. In a multi-threaded program we distinguish two roles that the threads may play.

- A thread running the TaskScheduler core loop plays the role of **SchedulerThread**. There can be at most one thread in this role, at any given time.
- All other threads are **ApplicationThreads**.

Ports are shared between ApplicationThreads and the SchedulerThread. ApplicationThreads may invoke methods from the Ports public API. For example, Send() and Recv() may cause the Ports internal state to go from Ready to NotReady. On the other hand, the SchedulerThread invokes io_WRITE() and io_READ() on the IoObject internal to the Port. This too may have the effect of altering the internal state of the Port. In addition the SchedulerThread invokes all the Port upcalls. In particular the ReadyHandler() implements the state transition back from NotReady to Ready.



The figure above illustrates how ApplicationThreads and the SchedulerThread may concorrently access a Port. In order to protect shared data within Ports we therefore need to synchronise thread-access. This challenge can be split into three sub-challenges.

1. Avoid race-condition the SchedulerThread and any ApplicationThread concurrently accessing a single Port.
2. Avoid race-conditions between any two ApplicationThreads concurrently accessing a single Port.
3. Avoid race-conditions between ApplicationThreads concurrently accessing different Ports.

We start by considering challenge 1); Race-conditions between an ApplicationThread and the SchedulerThread, within a single Port. At the first glace this does not appear to be much of a problem. This is because the Ready/NotReady protocol of the Ports already provides this kind of synchronisation. The state transition from Ready to NotReady in effect means that access privileges are yielded from the Application thread to the SchedulerThread. Similarly, the transition from NotReady back to Ready means that the SchedulerThread yields access privileges back to the ApplicationThread.

To be more precise even, the ApplicationThread yields to the SchedulerThread by first making the Port NotReady and then placing its internal IoObject in the ReadSet/WriteSet of the TaskScheduler. This may happen only after a partial read/write operation. The SchedulerThread now yields back to the ApplicationThread by following exactly the reverse sequence of actions. First the IoObject is removed from the ReadSet/WriteSet and then the Port is made to re-enter the Ready state. In essence, this is an instance of a token-based synchronising protocol between the two threads.

Unfortunately though, the Port implementation includes some ugly details that cause this otherwise elegant synchronisation protocol to be violated. First, the methods of the Connection API (e.g. SetConnection()) are available to both ApplicationThreads and the SchedulerThread, regardless of the Ready/NotReady state of the Port. This is not an accident though. The need to reset a Port or to replace its network connection may arise in various situations. In particular, a failed connection (Port is possibly NotReady) may need replacement and a working connection (Port possibly Ready) may simply not be needed any more.

So, this means that we have race-conditions after all, although not frequently. For instance, the SchedulerThread may be busy writing a message on behalf of a Port, while the ApplicationThread desides that the connection is no longer needed and invokes SetConnection(). Fortunately, the trivial solution to this problem seems appropriate.

*We require that only one thread can access a given Port at any time.*

This property is easily implemented using a standard locking mechanism. Each Port has one lock. This lock protects all access methods used by both ApplicationThreads and the SchedulerThread. This way, when a thread invokes an operation on a Port the lock is *acquired*. When the operation is completed the lock is *released*. Port ucalls are executed by the SchedulerThread without the lock being taken. Especially, this allows new Port operations to be initiated from the ReadyHandler().

This simple solution works well with Ports. First, Port operations are short, non-blocking operations. This means that if a thread ever blocks on the Port lock, the lock will be released shortly, possibly within the next slot of cpu-time given to the thread. Second, as demonstrated by the above discussion, race-conditions are rare in Ports, so this synchronisation is not associated with much overhead.

In addition, this effectively solves challenge 2) as well. Simultaneous access by multiple

ApplicationThreads is not possible with a single lock per Port. However, our implementation makes the assumption that there will not be fierce competition for Ports, between a large number of  ApplicationThreads. If this was the case a more sophisticated synchronization protocol might be useful. For example, a monitor-based [MON] implementation could make sure that busy-waiting on the lock was avoided and that the SchedulerThread was given priority.

This leaves us with challenge 3); Making sure there are no race-conditions between threads concurrently accessing different Ports. Fortunately, this is easy because Ports do not share much data. First, an InPort/OutPort pair may share the same socket. This works fine because the InPort only uses socket.read() whereas the OutPort only uses the socket.recv(). Second, all the Ports interact with the same TaskScheduler. Thus, in order to meet challenge 3) all we have to do is protect the data structures internal to the TaskScheduler. This includes the Timeout module, the ReadSet/WriteSet and the Task queue. Again, this is done with simple locks.

Finally, to enforce the assumption that there is only one ShedulerThread, the core loop of the TaskScheduler is protected by a lock of its own.

# 9. Implementing the Chain Proxy

The chain proxy is implemented as a Python module. The following steps are required to build and run a simple chain application. First, import the module and instantiate the chain proxy from the ChainProxyThread class. To start the proxy thread, invoke its start() method. Then invoke SessionStart(HeadIP) from the main thread to identify the chain and start a client-session. The application may now start sending Queries and Updates using the proxy. The client-session is ended by invoking SessionStop(). For more details, a simple example is available in file *apps/ProcMultiSynch.py.*

The chain proxy alone is about 1500 lines of python code. However, it is implemented on top of the Port abstraction, which itself is about 1700 lines of code. So, all included the chain proxy is about 3000 lines of python code. The code is not sparsingly commented though.

Our presentation of the chain proxy implementation is structured as follows. First we list the assumptions that we make about the implementation of the chain. Next we discuss the implementation of the session protocol, blocking operations and thread-safety. Finally we list remaining issues and potential improvements.

## 9.1 Assumptions about Chain Implementation

Before we discuss the implementation of the chain proxy in detail, it is helpful to list our assumptions concerning the chain implementation (i.e. the client-chain interface). We start by detailing how messages are transported across the network.

### 9.1.1 TCP

This implementation uses TCP to implement reliable, FIFO communication links between the chain proxy and the chain. TCP is a reliable byte-streaming protocol. There are several ways of implementing reliable message-streaming on top of reliable byte-streaming protocols. One way is to choose some special byte value as message delimiter. Another way is for all messages to include a fixed header containing the size of the trailing payload. The first approach has the unfortunate limitation that the payload can not include this special byte value. Since we do not want to enforce such a restriction on application programmers, we have implemented the second approach. Each message has a fixed-size header followed by a payload.

Communication endpoints are identified by an IP-address and a port-number. The Head of the chain accepts TCP connections from chain clients. Similarly, the chain client accepts TCP connection from the Tail of the chain. Both the Head and the chain client listen at well known ports. In our implementation the chain Head listens at port 5555 and the chain client listens at port 5556.

As discussed earlier, chain clients are assumed to know the initial IP-address of the Head. The Tail can connect to all chain clients using the contents of the replicated client-list (see the Client-Connect Protocol).

### 9.1.2 UDP

The chain client receives Udp-based Ping messages from the chain, so that the chain can detect a client failure. Ping messages are received at port 5557. The client will return the Ping unchanged to the (host, port) from which the Ping was sent. The chain client has only one requirment to the content of the Ping messages. The first four bytes must carry an integer (Big Endian)

### 9.1.3 TCP Message Structure

The chain proxy is concerned with multiple types of messages, i.e. Queries, Updates, Notifications and Replies. In addition the chain proxy and the chain must recognise the special meaning of messages associated with the Client-Connect Protocol, i.e. RegisterUpdates, UnregisterUpdates and their corresponding Replies. We have chosen to represent these additional messages as distinct message types, although they may well be understood as special cases of the before-mentioned messages. All in all this gives us the following list of message types.

1. Query
2. QueryReply
3. Update
4. UpdateReply
5. Notification
6. RegisterUpdate
7. RegisterUpdateReply
8. UnregisterUpdate
9. UnregisterUpdateReply

All messages are structured the same way. They have a header and a payload. The header includes information essential for the system. The payload carries application specific data, and it will typically include information such as objectID, operationID, parameter values or result values. The header is fixed size (28 bytes) and it is structured as follows.

| | | |
|---|---|---|
| [0-4] | int: | **LEN** (Big Endian) |
| [4-8] | int: | **ID** (Big Endian) |
| [8-12] | int: | **TYPE** (Big Endian) |
| [12-28] | string: | **CLIENT** (Ascii) |

**LEN** is the length of the entire message, including the header and the payload.

**ID** is a message identifier. Every chain proxy assigns a unique ID to every outgoing Update and Query message. These ID's are generated by incrementing a local message counter. In fact, the chain proxy keeps two such message counters, one for Updates and another for Queries. This solves several problems at once. First, message ID's makes it easy to associate Replies with Requests. When the chain generates a Reply for a given Update or Query, the request ID is passed on to the Reply. This allows the chain proxy to cancel timeouts for issued requests and it allows the client application to associate a given Reply with the correct

Reques. Second, the strict incremental ordering of these ID's makes it easy for the chain proxy to detect lost messages. This is why it makes sense to have two counters, one for Updates and one for Queries. Finally, note that lost messages that are retransmitted by the client application, will be assigned a new ID, like any other request.

**TYPE** is an integer refering to one of the message types listed above.

**CLIENT** is a string 16 byte representation of the IP-address of the chain proxy, e.g. "196.242.34.1". The actual address may be shorter than 16 bytes, so whitespaces should be stripped away. The Tail of the chain needs this information in order to set up new tail-connections. Also, when processing a Query or an Update the Tail needs to know which client should receive the Reply. This issue is solved by attaching the IP-address of the issuing chain-client on every request message.

With respect to Queries, Updates and Replies, the chain proxy does not make any assumptions concerning the content of the message payload. The payload is simply treated as a byte array. The content is only relevant once it is processed by a chain node (Query/Update payload), or when it is received by the application (Reply payload). For example, when a Query is processed by a chain node, the Query(query_string) operation of the local Storage Service is invoked with the message payload as parameter. Similarly, the result from this operation becomes the payload of the corresponding Reply message. So, the only requirement to message payloads of Queries and Updates is that they are understood by the Storage Services along the chain. In other words, the message payload must meet the requirements of the Chain-Storage-Interface. This however is entirely an application-level concern, so generally, the chain proxy need not care at all about payloads.

Notifications are exeptions to this rule. Rememer, Notifications are part of the private conversation between the chain proxy and the chain. They carry three pieces of information in the payload: Head-IP, Tail-IP and ChainLengt. The payload of Notifications is structured as follows:

```
[0-16]    string:  HEAD_IP (Ascii)
[16-32]  string:  TAIL_IP (Ascii)
[32-36]  string:  LENGTH (Big Endian)
```

### 9.1.4 Chain Behaviour

Finally, we summarize our assumptions concerning chain behavior.

- Head accepts TCP connections from clients, thereby establishing the Head-connection.
- Head receives all Update-types over the Head-connection.
- The chain recognises special messages RegisterUpdate and UnregisterUpdate.
- Tail always connects to the chain proxy, thereby establishing the Tail-connection.
- Tail always sends Notification as first message after connect.
- Tail accepts Queries on the Tail-connection.
- Tail sends all Reply-types and Notifications over the Tail-connection.
- Tail generates a Reply by giving it the same ID as the received Request.
- The chain is able to send Replies to the correct chain clients.
- The chain may occationally send Ping messages on both connections.
- The chain may end a client session by shutting down both connections.

As part of this project we have implemented a simple chain simulator. Essentially, it is a chain of length one, and it meets all the above requirements.

Finally, note that although our chain proxy is implemented in Python, we do not require that the chain is implemented in Python too.

## 9.2 Implementing the Session Protocol

In section 5 we defined the session-start protocol. In short it allows two connections, i.e. Head-connection and Tail-connection, to be established between a chain client and the chain. In addition the session-start protocol defines the state of a client-session as independent of the state of any of those two connections. Recall the steps taken to achieve this.

1. Chain client creates Head-connection.
2. Chain client sends RegisterUpdate over the Head-connection.
3. After RegisterUpdate is processed by Tail, Tail creates Tail-connection and sends a RegisterUpdateReply.
4. Chain client may start using the chain upon receipt of RegisterUpdateReply.

The session-stop protocol is very similar (when initiated by the chain client).

1. The client sends an UnregisterUpdate to the Head of the chain.
2. This request is then processed by the chain, causing the client to be safely removed from the replicated client-list.
3. The Tail generates an UnregisterUpdateReply and sends this over the Tail-connection.
4. The client receives the UnregisterUpdateReply and terminates both the Head-connection and the Tail-connection.

### 9.2.1 Session-Start Protocol and Session-Stop Protocol

The chain proxy implementation naturally focuses on the client-side of these two protocols, that is step 1,2 and 4 of the session-start protocol, and step 1 and 4 of the session-stop protocol. Fortunately, the implementation of these two protocols are similar. Especially, given that the implementation already provides mechanisms for sending Updates and receiving UpdateReplies, these mechanisms can be reused for special messages defined by the session protocol; Register, Unregister, RegisterReply and UnregisterReply.

For example, consider the method SessionStartAsynch(headIP)

```
def SessionStartAsynch(self, headIP):
        (state, res) = self._sendUpdate(chain.REGISTER, "", NONBLOCKING)
        if state != NOOP:
                # Nonblocking Connect to Head of Chain
                ChainProxyConnectObject(self, headIP, chain.CHAINPORT)
```

StartSessionAsynch() implements step 1) and 2) of the session-start protocol. This is reflected by the logic too which essentially only includes two statements. First, the method _sendUpdate() is used to create a Register request and commit it to the appropriate message port, i.e. the outHeadPort. If the port accepts the Register request (i.e. port was READY) the next step is to try to establish the Head-connection, given the

IP address of the Head of the chain.

Now, the sequencing of these two operations might seem counter-intuitive. Should we not first establish the connection and then send the message? The problem with this approach is that we want the entire operation to be non-blocking. That is, we do not want to wait for sock.connect() to complete before we return from the method invocation. Fortunately, the port abstraction gives us a much better solution. Recall that an OutPort may accept a message, even if it is not set up with a proper connection. It can do this because is has a buffer space of a single message. Now, what happens next is that sock.connect() completes asynchronously, some time *after* the application has returned from StartSessionAsynch(). At this point, the socket is writeable and may safely be given to the outHeadPort by using SetConnection(). The outHeadPort will thus be able to send the Register request directly after sock.connect() the Head-connection became writeable, which is exactly what we want.

SessionStartAsynch() also supports repeated invokation, in the event that the previous connect fails. The chain proxy recognises this by noticing that the outHeadPort is NOT READY (busy sending the previous Register request). In this case it is not neccassary to create a new Register request. Instead, the implementation skips directly to a second attempt at establishing the Head-connection.

After this, the rest of the protocol is fairly simple. The chain proxy accepts connections from the Tail, and when the RegisterReply arrives the session is said to be live. SessionStopAsynch() is simpler than SessionStartAsynch(), because it does not request any connections to be established. When UnregisterReply is received, the session is no longer live, and both Head-connection and Tail-connection are closed.

### 9.2.2 Local Session View

The session-start protocol goes through several steps before the client session is said to be live. Likewise, multiple steps are required to end a client session. In addition, fails or events may occur during these transition that may require the protocol to be aborted. In order to deal with this complexity the chain proxy maintains a centralised session object. All protocol steps interact with this session object in order to keep it up to date. This way it is possible to keep a consistent local view of the session-status.

The session object is useful in many ways.

- It is easy to implement the SessionStatus() operation mentioned in section 5. This is simply a query to the session object.
- The implementation of the session-start protocol and the session-stop protocol may also query the session object, in order to make informed decisions.
- The state of the session object is used to protect the chain proxy from out-of-protocol usage by the application. Especially, it is not possible to send a Query or an Update, unless the session objects says that the client-session is live. Conversely, it is not possible to start a new client session unless the previous client session has been properly terminated.
- The state of the session object is used to protect the chain proxy from out-of-protocol usage by the chain. This involves accepting incoming connections from the chain only when they are expected, i.e. at any time after the session-start protocol has been initiated, until the session-stop protocol has been finalised. To see why this may be useful, consider the following scenario. The chain proxy has set a small value for SessionTimeout. Now, if both Head and Tail fails simultaneously the chain might not be able to reconnect to the chain proxy, before the session timeout expires at the chain proxy. In this case the chain learns that the client has ended the session, by not being able to reconnect.

- Similarly, the chain proxy will stop responding to Ping messages once the client session is terminated. This too helps the chain realise that the chain proxy has aborted the session.

### 9.2.3 Complete Chain Failure or Session Terminated by Chain

In section 5 we also discussed client behavior in the event that the chain completely fails, or in the event that the chain shuts down both connections (and never re-connects from Tail) for other reasons. In this case there is not much to do for the chain proxy, except waiting a while before terminating the client session. The value of SessionTimeout defines the length of this interval. The chain proxy achieves this behaviour by registering a timeout every time both Head-connection and Tail-connection is down. Whenever a new Tail-connection is established, the timeout is cancelled. If not, the client-session is terminated locally, and the SessionTimeoutHandler() is invoked.

The details of the session-start and session-stop protocols are studied in detail in section 11.

## 9.3 Implementing Blocking (Synchronous) Operations

The Chain API includes a blocking (synchronous) version of its four principal operations;

*SessionStart(), Query(), Update() and SessionStop()*

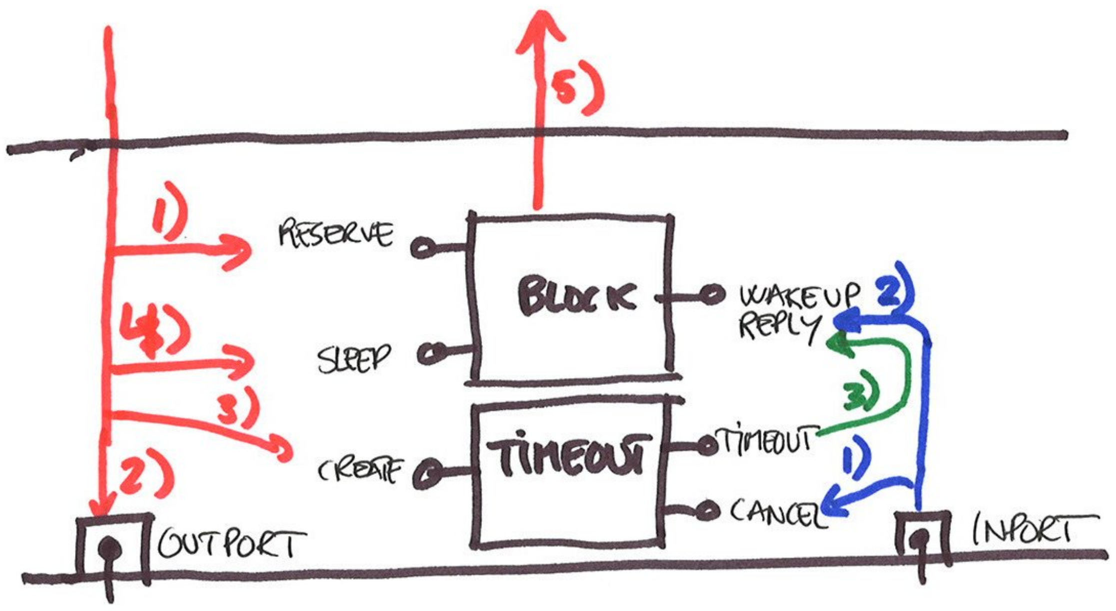
The semantics of these blocking operations are both block-on-reply and block-on-send.

- Block-on-reply, if a request was successfully sent, the application thread will be blocked until the Reply is available, or until the Request is timed out. The return value will indicate timeout (TIMEOUT) if that is the case.
- Block-on-send, if a request could not be completely sent immediately, the operation will block-on-send. The block-on-send behaviour depends on the value of SendTimeout, as discussed in section 7.

However, blocking behaviour is implemented by a single mechanism, independent of the reason for blocking. This mechanism was first implemented as a simple Wait-Wakeup protocol. Once the application thread had been able to issue a Request it went to sleep on an python threading Event, associated with the RequestID. Then as the appropriate Reply was received by the proxy, or as some timeout expired, the appropriate result value was given to the appropriate sleeping thread, just before the Event flag was set - causing it to wake up.

This solution though is not entirely safe. Consider the following scenario. The ApplicationThread manages to send a Query (completely), but is interrupted by a thread-switch just before it could go to sleep. Then the SchedulerThread will run for a while, and possibly it could receive the Reply (if the network RRT < thread cpu time-slot). In this situation, the SchedulerThread would not find a sleeping application Thread to wake up, so the Reply is lost. Event worse, when the ApplicationThread would then go to sleep as soon as it is re-scheduled, and will never be woke up again.

Admittedly, this scenario is not very realistic across a WAN or even a LAN. However, we have observed this failure when running both the proxy and the Tail of the chain on the same physical machine. Fortunately this problem is easily avoided. We just added another step to the protocol. It is now a Intention-Wait-Wakeup protocol. The Application Thread will register its intention to send a Request prior to actually sending it. This way, if the Reply is received before the Application Thread was able to go to sleep, the Scheduler Thread will recognize the intention and store the result value. Then instead of going directly to sleep, the Application Thread always checks to see if the Reply is already available, in which case it can return directly without going to sleep at all.



The figure illustrates the steps required to implement block-on-reply. 1) the thread reserves a place in the "block" module. 2) the request is sent. 3) the timeout is created. 4) the Application Thread goes to sleep. At some point later, the SchedulerThread will either 1) receive the Reply or 3) receive a timeout. Then in 2) the Application Thread is awoken with either the Reply or the Timeout value. Finally in 5) the Application Thread returns from the blocking invocation.

The implementation of block-on-send only slightly complicates this. The SendTimeout is Registered just before the request is issued to the Port, and cancelled as soon as the Port has completed writing the request to the network. If this is not soon enough, the SendTimeout expires in exactly the same way as the ReplyTimeout.

As a rather obvious warning. Single-threaded programs can not use the blocking Chain API. This will cause the program to deadlock, since there will be no thread to run the TaskScheduler once the Application Thread has gone to sleep.

## 9.4 Thread-Safety of the Chain Proxy

In ensuring the thread-safety of the chain proxy, the thread-safety of the internal Message Ports helps a lot. Recall, Ports are synchronized so that they can be accessed only by a single thread at the time. Still, there are several other issues that need to be resolved as well.

**Issue 1 : Atomicity of Query and Update operations.**

If two application threads, A and B, are both trying to send a Query at the same time, this constitutes a potential race-condition. To see the problem, consider this pseudo-code version of the QueryAsynch() method.

```
    QueryAsynch(query)
1)        If OutTailPort.IsReady()
2)              queryID = NextQueryID()
3)              CreateTimeout(queryID)
4)              OutTailPort.Send(queryID, query)
```

Say thread A is interrupted after having executed steps 1), 2) and 3), but before actually having sent the Query. Thread B is then scheduled. It will conclude that the Port is still ready, and proceed. As a consequence, thread B will get the next queryID and send its Query. Now, what happens next will depend on the state of the OutTailPort, after B's send operation has completed.

Either, the OutTailPort might be Ready for the next Query. Then, if thread A is scheduled afterwards, it too will be able to send its Query. The result is that both Queries are sent successfully, but in the wrong order. (The correct order is detemined by the queryID's). This violates a central assumption of the chain replication protocol.

Or, the OutTailPort might be NotReady. In this case thread A will fail sending its Query. For the application this will not be different from being refused at step 1), so it is not a really a problem. However, a queryID has been pulled for which no message has been sent. This will be interpreted as a Lost message by the chain proxy. If not catastrophic, but it will at least seem puzzling to the application programmer when the TimeoutHandler() is invoked with an unrecognized queryID.

Fortunately, both these outcomes can be avoided by making sure the QueryAsynch operation is atomic. A simple lock around this critical section will do the trick. This goes for the other operations as well.

**Issue 2 : Independence of Query-stream and Update-stream.**

As we just discussed, two threads competing for the Query operation will have to be sequentially synchronized. However, if thread A attempts to send a Query while thread B attempts to send an Update, there should be no reason to block either. In fact, we especially want Query operations and Update operations to be independent as much as possible. To achieve this we distinguish between a Query-lock and an Update-lock. The Query-lock is associated with the Query operation, while the Update-lock is shared between three operations; Update, SessionStart and SessionStop.

**Issue 3 : Shared data structures between Update-stream and Query-stream.**

At this point, two application threads may run simultaneously within the proxy. One is sending a Query and one is sending an Update. Data that is updated by both these threads needs to be protected from race-conditions. We address this by avoiding shared data as much as we can. Datastructures needed to manage the two streams, i.e. counters and timeout-maps are duplicated for separation. However, there is one exception. The scheduler offers a TimeoutService where a thread may register a timeout. This too could be duplicated, however, since the TimeoutModule was already thread-safe, we decided it would not worth the effort.

**Issue 4 : Race conditions between TaskScheduler and Application-threads.**

The chain proxy is run by the TaskScheduler of the Port abstraction, quite possibly running in a separate thread. We call it the proxy-thread. The Ports are protected from race-conditions between this proxy-thread and the application-threads. Still, the proxy-thread reaches outside the Port whenever an upcall is invoked. The Port handlers are therefore another source of race-conditions. This leads us to protect all internal data structures that are shared between any of the Request-streams and the Reply-stream. Fortunately, this is only the logic associated with managing timeouts and blocking threads.

**Issue 5 : Blocking Query and Update Operations.**

In issue 1) we discussed how atomicity of Query and Update operations can be ensured using locks. However, with regard to blocking operations, it is crucial that the application thread releases the lock before it blocks. A common pattern is to release the lock before going to sleep, and then grabbing it again as soon as thread wakes up. This last step will not be necessary in our case, because the blocked threads do not touch shared data after having been awoken.

**Issue 6. Upcalls and Thread Switching.**

All upcalls of the chain proxy are invoked by the proxy-thread. Given that the chain proxy is executed in a separate thread from the application-thread, the application programmer must be aware of this source of thread-related issues. S/he may deal with this in several ways. If the upcalls are allowed to touch application data, this raises the potential for race conditions within the application. Shared data in the application must then be protected. Another alternative is to perform a thread-handover. For example, by handing the Reply over, from the proxy-thread to the application-thread using a thread-safe buffer.

In any case, it is always safe to invoke new operations on the chain proxy from within an upcall-handler. Note however that the upcalls are supposed to be non-blocking and light-weigth. So, it might not be a good idea to send large set of requests back-to-back, from within a single upcall. It is indeed possible, but as long as all the operations succeed the entire task may continue for a long time. In this time the chain proxy will be unresponsive to anything else, including new Replies becoming available or expiring timeouts. This is because all this work will be executed within a single Task, by the TaskScheduler within the chain proxy. Not until this Task is completed will the chain proxy be able to do anything else. Incidentally, this is a prime example of how the application may (unintentionally) flood the chain with Request while not receiving any Replies.

**Issue 7. Recv before Send also an issue with Non-Blocking operations.**

In the subsection of blocking operations, within section 7, we discussed a particular problem with thread-switches and blocking threads. In short, it appeared that a Reply could be received before the Request was sent. This (illusion) was possible if a thread-switch occurred at an unfortunate point, just after the Request was sent, but before the application thread was able to return from the operation. For blocking operations we had to fix this problem, since it caused deadlocks.

Unfortunately, this problem applies to non-blocking operations as well, though still in the same peculiar circumstances. Essentially the round-trip-time of a Query must about the same length as a thread-scheduling slot. We have observed this only when testing against a chain simulator running at the same host.

In any case, there is no simple way around this, so the issue will have to be resolved by the application programmer. Fortunately, this should not be hard. For example, an application
might keep track of pending requests using a service that supports two operations; pend(id) and unpend(id). In order to solve this problem this service only need to be improved so that it support invocations of pend() and unpend() in the non-intuitive order.

## 9.5 Remaing Issues and Further Improvements

The implementation of the chain proxy is quite complete. Still, there remains some potential for further improvement. Here we list a couple of issues where the chain proxy could be made more robust or more effective.

- Improved efficiency of the Port implementations. The logic of the Port implementation that deals with reading and writing message to sockets is so far inefficient. It should be reimplemented, possibly in C, to reduce the overhead as much as possible.
- Another issue with the implementation of Ports relates to large messages. For the moment the port abstraction may read or write huge messages in a single Task in the TaskScheduler. This will make the chain proxy rather unresponsive during transfer of large messages. This breakes the assumption that Tasks should be small, non-blocking operations. Thus, it would be better if large messages were read and written by a series of Tasks rather than in a single Task.
- We mentioned earlier that the Timeout Service of the TaskScheduler could be duplicated to reduce interference between two thread simultaneously sending a Query and an Update. This is not difficult, but the gain from this is probably very small.

# 10. Application Development

This section discusses various topics related to development of chain applications. First we give some mock-up examples of various types of chain applications. Next we discuss some important issues related to application design. Finally, we present a real application, a naive implementation of a chain-replicated file system.

## 10.1 Application Types

The chain proxy is designed to support various application types. Applications may be single-threaded or multi-threaded and they may be expressed in event-based or procedural form[8] . In addition, we may distinguish applications in that they use blocking (synch) or non-blocking (asynch) operations, or even a mixture. Here we present the most likely combinations of the features just listed.

**Event-based/Single-threaded/Asynchronous**

The chain client is a single-threaded program and the application logic is expressed in an event-based manner. There is a single event-loop in the program, running both the application and the proxy. One way to achieve this is to implement the entire application as a collection of handlers and bind their execution to upcalls from the proxy. After the application has been started with SessionStart() the infinite Run() method may be called. The rest of the application will execute as a cascading series of reactions to events. A single-threaded program implies that only asynchronous (non-blocking) operations may be used. An example implementation of such a chain client is to be found in:

*apps/EventSingleAsynch.py*

**Procedural/Single-threaded/Asynchronous**

The chain client is a single-threaded program, yet the application logic is expressed in a procedural manner. This is possible if the program statements of the application are manually interleaved with invocations of RunOnce() of the proxy. A single-threaded program again implies that only asynchronous (non-blocking) operations may be used. An example implementation of such a chain client is to be found in:

*apps/ProcSingleAsynch.py*

**Event-based/Multi-threaded/***

The chain client may for instance have two threads. One thread is running the proxy event-loop, i.e. Run() and the other is running the application event-loop. The application may bind the execution of its handlers both to proxy events and application events. The non-blocking nature of event-based programs should imply that only asynchronous (non-blocking) operations will be used. However, blocking operations may be used too, at least by one of the threads. An example of such a chain client is discussed at the end of this section, and the code example is available in:

*crfs/crfsClient.py*

8. Receiving Replies can by default not be done in a Procedural manner, since Replies are always delivered by the ReplyHandler(). However, the application can implement Reply-buffering, thereby allowing a Recv() method to be added. This would allow Replies to be received in a Procedural manner as well.

**Procedural/Multi-threaded/Synchronous**

The chain client may for instance have two threads. One thread is running the proxy and the other is running the application. The application may then use synchronous (blocking) operations exclusively. For instance, this may be attractive for simple terminal-based, interactive chain clients. An example is to be found in:
*apps/ProcMultiSynch.py*

**Procedural/Multi-threaded/Asynchronous**

The chain client may for instance have two threads. One thread is running the proxy and the other is running the application. The application may then use asynchronous (non-blocking) operations exclusively. This would allow the application to take advantage of the pipelining capabilities of the chain. The application might have to do some waiting whenever the proxy is not ready to accept new requests. An example implementation can be found in:
*apps/ProcMultiAsynch.py*

## 10.2 Programming Chain Applications : Some General Issues

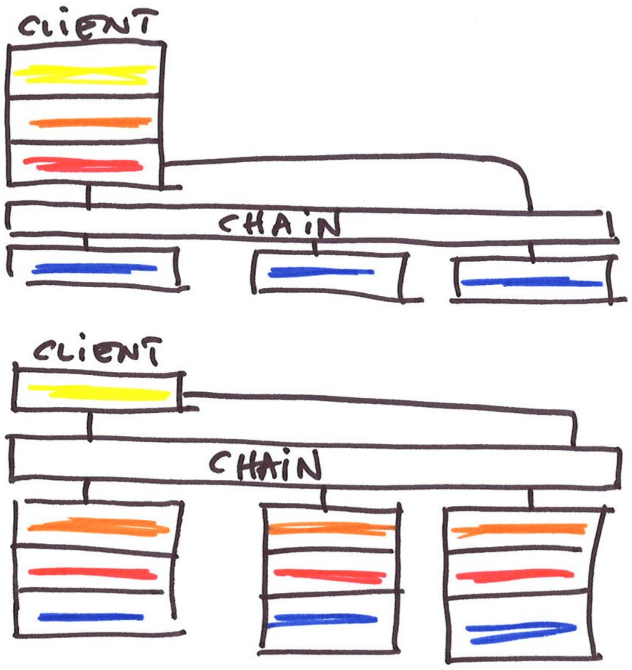
Here follows a couple of issues that may apply to the design of many chain applications.

### 10.2.1 Mapping Storage Interaction to Queries and Updates

In order to use chain replication, applications need to express their storage interactions in terms of Queries and Updates. Functionally, this is usually not a problem, even for highly sophisticated data-structures. The reason is that Queries and Updates can always implement the most basic storage abstractions, for instance a key-value mapping with Lookup/Insert/Delete primitives. More complex data-structures can then be built on top of this. So, if the application depends on some high-level storage API, it is always possible to wedge in a software layer on the client that translates between high-level and low-level primitives. This way, a chain that replicates primitive data objects, such as key-value pairs, could be used by a wide variety of applications.

### 10.2.2 Granularity of Operations

On the other hand, if possible, there may be reasons to implement replication at a higher level of abstraction. In essence, this may have the effect of packaging a sequence of low-level operations into a single high-level operation. One motivation for doing so would be to reduce latency due to dependencies between operations. For example, consider two Update operations U1 and U2, where U2 in some way depends on the outcome of U1. Due to this dependency, the two Updates can not be pipelined by the application, so the latency of the whole operation is two times the expected Update-latency of the chain. If instead U1 and U2 can be combined into a single new Update U3, the latency of the whole operation can be cut in half. The fundamental trick here is to resolve the dependency within the chain (close to the data) rather than at the client (far from the data). Replication of such higher-level objects generally takes advantage of the processing capabilities within the chain, while at the same time reducing the complexity of the client.



The above figure illustrates low-level and high-level replication, respectively. A software stack is divided between the client application and the chain nodes. Replicating middleware (i.e. the chain proxy and the chain) can in principle be inserted between any two layers in this stack.

Unfortunately, in order to make this kind of choice the application programmer will most likely need administrative control over the chain. At least, there must be a mechanism that allows the programmer to define a custom implementation of the storage service (internal to each node in the chain). This way, the application programmer could develop the chain application and the storage service in tandem. Incidentally, the chain itself might be an excellent basis for such a mechanism. For example, the code that runs the storage service could also be made available as an ordinary, replicated object. If this was the case, a new custom implementation of the storage service could be deployed simply by sending an Update.

### 10.2.3 Request Scheduling

Now, whether the storage API is open to modification or not, there is still much that needs to be done by the application programmer, in order to make good use of the pipelining capabilities of the chain. First, the application programmer has to split the stream of storage requests into two streams, Queries and Updates. This may or may not be difficult, depending on the application. One challenge especially is to avoid dependencies between those two streams. If Updates must wait for Queries to complete, and visa versa, this increases latency and reduces throughput. The same is true for dependencies internal to each stream, as discussed above.

Such challenges are well known from pipelining literature and solutions may be inspired by research in this area. However, in chain replication, scheduling of Queries and Updates is not simply a matter of sending them into the pipelines in the correct order. This is because the application programmer does not fully control the order of Queries relative to Updates. Rather, the ordering is decided by the Tail of the chain. Instead, request scheduling policies on the client need to be sensitive to the order in which Replies arrive. This is how the chain client learns about the actual ordering of requests in the system.

### 10.2.4 Rewriting Applications

Applications do not necessarily need to be rewritten in order to make use of chain replication. The choice of replicating middleware could be made completely transient to the application, for instance by implementing a chain-replicated file system and mounting this into the local file system, as is demonstrated below. Note however that this may not always be optimal considering that 1) assumptions made by the application concerning relative latency of read and write operations may be violated, and 2) the application does not necessarily make the best use the pipelining ability of chain replication.

### 10.2.5 Well-Suited Applications

Finally, some applications especially are very good candidates for chain replication. These are  applications where the Query-stream and the Update-stream are logically independent. The OSDI paper [CR] uses applications such as large scale storage services and search engines as motivation. In these applications Queries and Updates typically originate from different processes. For instance, an indexer-process would issue Updates to the search-index, while Queries are issued by remote end-users.

## 10.3 CRFS: Chain Replicated File System

Here we present a simple implementation of *crfs*, a reliable file system based on chain replication. The purpose is to indicate one specific utility of the chain proxy.

Any implementation of a file system will have to implement a well known set of system calls, including file operations, directory operations and more general operations. For example, mknod, open, read, write, rename, mkdir, readdir, rmdir, link, unlink, stat, etc. So, when designing a chain-replicated file system the basic challenge is translating all these syscalls into sequences of Queries and Updates. We have chosen a very primitive solution for this. The file system operations are simply divided into two groups, *Query-Operations* and *Update-Operations*, based on wheter the operation changes the state of the file system or not.

**Query Operations**
getattr, read, readdir, readlink, statfs

**Update Operations**
open, write, truncate, rename, release, mknod, mkdir, rmdir, link, symlink, unlink, chmod, chown, utime, fsync

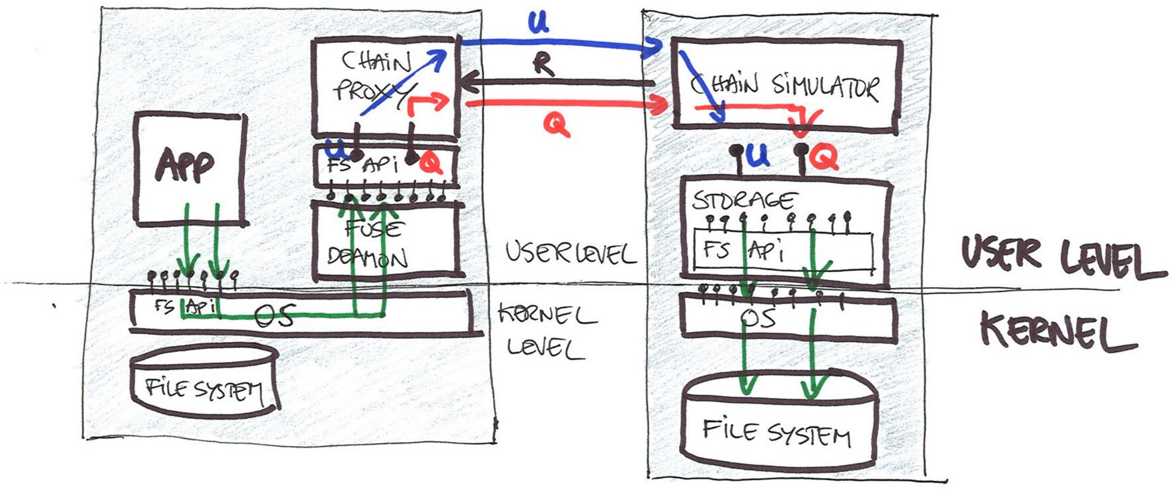
This way, all file system operations may be mapped directly to a blocking invocation of Query(*request_string*) or Update(*request_string*) on the chain proxy. The idea is to implement this similar to RPC[RPC]. The chain client may construct each *request_string* by serializing the name of the file system operation along with the given parameters. Chain nodes may then extract this information and execute the appropriate file system operation, towards the local file system implementation. The result value is then included in the Reply and carried back to the chain client.

The idea is simple, yet implementing such a custom file system need not be. Generally file system implementation is non-trivial because it involves modifications to the operation system kernel. Fortunately though Fuse[FUS] makes this much easier by allowing custom file systems to be implemented as user-level processes. Fuse provides a kernel extension that intercepts system calls targeted to specified parts of the file system. These system calls are then represented as a task, and the responsibility for executing this task is handed over to a user-level process. The user-level process is continuously polling for such tasks, and will invoke the appropriate handler whenever a new task is detected. So, with Fuse, implementing a custom file system is simply a matter of implementing these handlers

In our solution, the Fuse user-level file system is implemented by integrating it with a chain proxy. A thin layer on top of the chain proxy implements the mapping from file system operations to Queries and Updates, as described above. The entire process (both a user-level file system and a chain client - depending on perspective) includes two threads. One thread runs the event-loop of the chain proxy, while the other runs the event-loop of the Fuse deamon. The Fuse deamon makes blocking calls on the proxy Threads. With reference to the above taxonomy, this setup would be labelled *Event-based/Multi-threaded/Synchronous*.

The chain simulator is set up with a specific storage object that meets the requirement indicated above. Recall, the storage object must extract information from the request in order to execute the file system operations locally. As discussed in section 4, there must be a direct correspondence between the *application-chain interface* and the *chain-*

*storage interface*. In our implementation these are both the same file system interface dictated by the Fuse implementation.



The figure illustrates how a chain-replicated storage system may be made available for any application that uses the local file system.

The implementation of CRFS includes two files. CrfsClient.py requires that Fuse is previously installed along with fuse-python-bindings

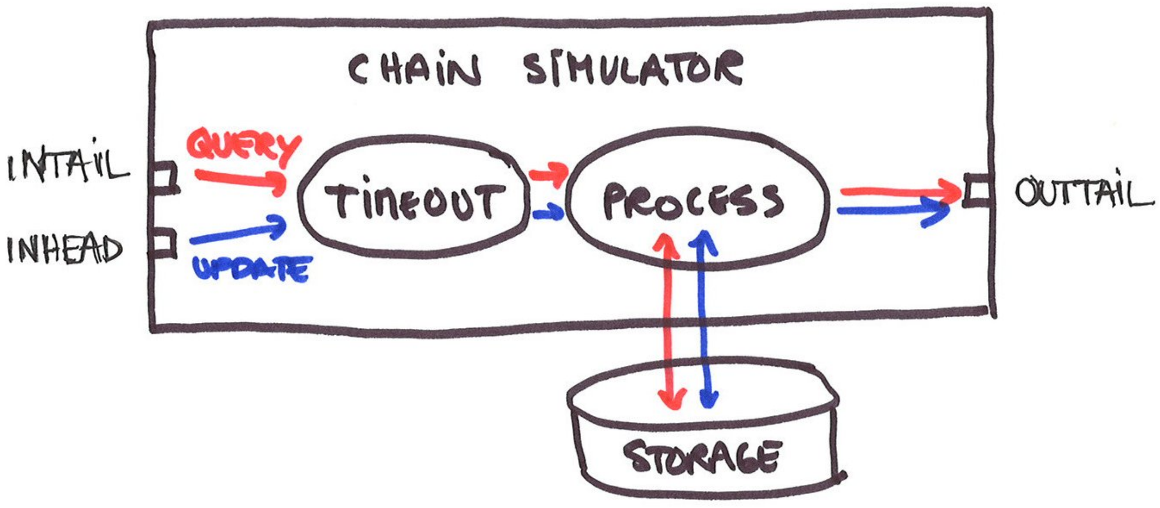
- *crfs/crfsClient.py* : implements the fuse-deamon/chain-client.
- *crfs/crfsChainsimulator.py* : sets up a chain simulator with a custom storage object.

# 11. Evaluation

This section presents the evaluation of the chain proxy implementation. First we present the implementation of the chain simulator, which is used throughout the experiments. Then we go on to evaluate functional correctness. This addresses the correctness of the session protocol in particular. Finally we present the performance characteristics of the chain proxy, under normal execution and in the event of chain failures.

## 11.1 Chain Simulator

The chain simulator is similar to a chain of length one, i.e. a single node playing the role of both Head and Tail simultaneously. From the perspective of the chain proxy, it is supposed to be indistinguishable from a real chain. This is achieved by implementing the client-chain interface discussed in section 9. So, the chain simulator maintains two connections, a Head-connection and a Tail-connection. Updates and Queries are received from the chain proxy, and the application-dependent operations they include are applied to a local storage object. Results are sent back to the chain proxy as Replies on the Tail-connection. In addition, the chain simulator implements the session protocol.



The above figure illustrates the request/reply stream through the chain simulator.

Still, the chain simulator is different from a single-node chain in a couple of respects. First, it is able to fake being a chain of a certain length. This is done simply by delaying Queries and Updates, before they are processed, in order to simulate Query processing costs and Update propagation delay. The Update propagation delay is set to be proportional to the length of chain. Second, the chain simulator simulates chain failures. An API is provided that allows typical chain failures to be triggered within the chain simulator. The list of failures closely mirror the failure scenarios discussed in section 3.

- **HeadDown**. Head-failure is simulated by closing the Head-connection. Chain length > 1
- **HeadUp**. Chain has recovered with a new Head. Chain length decreases. A notifiaction is sent to the chain proxy.
- **TailDown**. Tail-failure is simulated by closing the Tail-connection. Chain length > 1

- **TailUp**. Chain has recovered with a new Tail. Chain length decreases. A new Tail-connection is established and a notification is sent to the chain proxy.
- **MiddleDown**. Failure of intermediate node. Noop.
- **MiddleUp**. Chain has recovered from intermediate failure. Chain length decreases. Notification is sent to the chain proxy.
- **HeadTailDown**. Simultanous Head and Tail failures are simulated by simultaneously closing both Head-connection and Tail-connection. Chain length > 2.
- **HeadTailUp**. Chain has recovered from simultaneous Head and Tail failures. Chain length decreases by 2. Tail-connection is established and a notification is sent to the chain proxy.
- **Extend**. Chain is extended with a new node. Chain length increases. This is simulated by closing the Tail-connection and then reestablishing it. A notification is sent to the chain proxy.
- **DropUpdate**. Chain drops the next Update before it is processed.
- **DropQuery**. Chain drops the next Query before it is processed.
- **DropQueryReply**. Chain drops the next QueryReply before it is sent.
- **DropUpdateReply**. Chain drops the next UpdateReply before it is sent.

The implementation of the chain simulator is based on our implementation of Message Ports. This is fortunate because the Port abstraction simplifies the task considerably. All the complexity with handling messages and connections is handled by the Ports. In addition, the Timeout Service within the TaskScheduler is a well suited mechanism for delaying Updates.

The code of chain simulator is to be found in */chain/chainSimulator.py*. In addition, a console-based interface to the chain simulator is provided in */chain/consoleSimulator.py*. This allows access to the chain simulator from the command line. This way, failures may be triggered manually while the chain simulator is running.

## 11.2 Demonstrating Correctness

The chain proxy have a number of functional aspects that we would like to test for correctness. For example, the session protocol, Query/Update processing, Query/Update independence, message loss, timeouts, reactions to failures and recovery. For this purpose we have developed a console-based interface to the chain proxy as well as the chain simulator. This way, by carefully controlling the state and the steps taken by both chain proxy and chain simulator, we may easily test a host of scenarios. We have done this kind of testing throughout the development process, and at present we do not know of any bugs related to central funtionality.

The storage object used by the chain simulator in all these experiments is trivial. As illustrated by the below figure, it keeps a simple value (string). Updates simply replace this string, and Queries return a copy (regardless of parameter).

```
class Storage:

    def __init__(self):
        self._message = ""

    def Query(self, query_str):
        return self._message

    def Update(self, update_str):
        self._message = update_str
        return "ok"
```

This evaluation of functional correctness has two parts. First we test the core functionality of the chain proxy, i.e. the Query() and Update() primitives. Next we focus on the correctness of the session protocol, i.e. the SessionStart() and SessionStop() primitives. For each primitive we will test both the synchronous and the asynchronous variants. In addition we will test all primitives with respect to two kinds of failures, Head-failure and Tail-failure.

## 11.3 Synchronous Queries and Updates with Chain Failures

In this experiment we study how synchronous Query and Update operations are affected by failures in the chain. Below follows an outline of the experiment. ">>>" is the python interpreter prompt. Each steps is given a number at the left and will be commented below. The console interface to the chain proxy is implemented in */proxy/consoleClient.py* whereas the console interface to the chain simulator is found in */chain/consoleSimulator.py*. The chain proxy is running in *Procedural Mode* (see section 6).

| | **python consoleClient.py** | **python consoleSimulator.py** |
|---|---|---|
| | >>> | >>> |
| 1) | >>> **SessionStart**("headIP") | |
| | ('OK', '') | |
| 2) | >>> **Update**("Value1") | |
| | ('OK', 'ok') | |
| 3) | >>> **Query**("") | |
| | ('OK', 'Value1') | |
| 4) | | >>> **HeadDown**() |
| 5) | >>> **Query**("") | |
| | ('OK', 'Value1') | |
| 6) | >>> **Update**("Value2") | |
| 7) | | >>> **HeadUp**() |
| | ... ChainLengthHandler(2) | |
| | ... UpdateReadyHandler() | |
| | ('OK', 'ok') | |
| 8) | **Query**("") | |
| | ('OK', 'Value2') | |
| 9) | | >>> **TailDown**() |
| 10) | **Update**("Value3") | |
| | ('TIMEOUT', None) | |
| 11) | **Query**("") | |
| 12) | | >>> **TailUp**() |
| | ... QueryReadyHandler() | |
| | ... ChainLengthHandler(1) | |
| | ('OK', 'Value3') | |
| 13) | **SessionStop**() | |
| | ('OK', '') | |

**1).** The session start protocol is started. The chain proxy establishes the Head-connection to the chain simulator and sends a RegisterRequest. The chain simulator then establishes the Tail-connection and sends back a Notification and the RegisterReply. The method invokation blocks until the Reply is received.

**2).** An Update is sent via the Head-connection and processed by the chain simulator. The blocking method returns successfully as the UpdateReply is received.

**3).** A Query is sent via the Tail-connection and processed by the chain simulator. The blocking method returns the recently updated value of the storage object, as soon as the QueryReply is received.

**4).** The Head-connection is taken down by the chain simulator.

**5).** A new Query is issued by the chain proxy. This succeeds since the Tail-connection is not affected by Head-failure.

**6).** A new Update is issued by the chain proxy. However, since the Head-connection is down, the Update can not be sent. Update() is a blocking method, i.e. blocks until the UpdateReply is available. Since no SendTimeout is specified the chain proxy, the invokation of the Update() method will block indefinitely.

**7).** The chain simulator notifies the chain proxy that a new Head is ready, and that the chain length has changed. This causes the ChainLengthHandler to be invoked. The chain proxy will then re-establish the Head-connection. This causes the Update from 6) to be sent. As a consequence, the chain proxy now is ready to accept a new Update. This is signalled to the application by invoking UpdateReadyHandler(). Finally, as the UpdateReply is received, the Update operation from step 6) may finally unblock, reporting that the Update has been successfully applied to the chain.

**8).** A new Query() confirms that this is indeed the case.

**9).** The chain simulator shuts down the Tail-connection.

**10).** The chain proxy issues a new Update. Since the Head-connection is live, the Update is sent and then processed by the chain. However, since the Tail-connection is down, the Reply can not be sent. As a consequence the Update eventually times out.

**11).** The chain proxy now issues a Query to check whether the previous Update was successful. The Query method will block since the Tail-connection is down.

**12).** The chain simulator reconnects to the chain proxy. This immediately causes the Query from 11) to be sent across the new Tail-connection and then QueryReadyhandler() to be invoked by the chain proxy. Next, the chain proxy receives a new notification with a fresh chain view. This causes the ChainLengthHandler() to be invoked. Finally, when the QueryReply arrives, the Query operation from step 11) unblocks and returns the correct result.

**13).** Finally, the session is ended by running the session stop protocol. The chain proxy sends a UnregisterRequest to the chain and waits until the corresponding UnregisterReply is returned. At this point both the Head-connection and the Tail-connection is closed by the chain proxy.

## 11.4 Asynchronous Queries and Updates with Chain Failures

We repeat the same experiment, only this time with asynchronous operations exclusively.

| | | |
|---|---|---|
| 1) | >>> **SessionStartAsynch**("headIP") | |
| | ('INIT', 'U1') | |
| | ... SessionStartHandler() | |
| 2) | >>> **UpdateAsynch**("Value1") | |
| | ('OK', 'U2') | |
| | ... UpdateReplyHandler(U2, ok) | |
| 3) | >>> **QueryAsynch**("") | |
| | ('OK','Q1') | |
| | ... QueryReplyHandler(Q1, Value1) | |
| 4) | | >>> **HeadDown**() |
| 5) | >>> **QueryAsynch**("") | |
| | ('OK','Q2') | |
| | ... QueryReplyHandler(Q2, Value1) | |
| 6) | >>> **UpdateAsynch**("Value2") | |
| | ('INIT', 'U3') | |
| 7) | | >>> **HeadUp**() |
| | ... ChainLengthHandler(2) | |
| | ... UpdateReadyHandler() | |
| | ... UpdateReplyHandler(U3, ok) | |
| 8) | >>> **QueryAsynch**("") | |
| | ('OK', 'Q3') | |
| | ... QueryReplyHandler(Q3, Value2) | |
| 9) | | >>> **TailDown**() |
| 10) | >>> **UpdateAsynch**("Value3") | |
| | ('OK, 'U4') | |
| | ... TimeoutHandler(U) | |
| 11) | >>> **QueryAsynch**("") | |
| | ('INIT', 'Q4') | |
| 12) | | >>> **TailUp**() |
| | ... QueryReadyHandler() | |
| | ... ChainLengthHandler(1) | |
| | ... QueryReplyHandler(Q4, Value3) | |
| 13) | >>> **SessionStopAsynch**() | |
| | ('OK', 'U5') | |
| | ... SessionStopHandler() | |

Since this is the same experiment as above we will limit the discussion to comment on the most interesting steps.

**1).** SessionStartAsynch() returns immediately ('INIT', 'U1'). This indicates that the Register request is identified by U1, and that it could not be completely transmittet before the operation returned. This is to be expected, because the invocation of SessionStartAsynch does not wait for the Head-connection to be connected. Instead, the Register will be sent at some point later, as soon as the Head-connection is operational. Normally this would cause UpdateReadyHandler() to be invoked, to signal that the chain proxy is ready to send the next Update. This however does not happen here. This is because the client session is not established yet, and since no Updates or Queries can be accepted out of session, it seems pointless to signal readyness. So, the chain proxy only invokes UpdateReadyHandler() and QueryReadyHandler() while the client-session is live. Note also that the UpdateReplyHandler() is not invoked for RegisterReplies. Instead, SessionStartHandler() is invoked in its place.

**2).** UpdateAsynch() sends an Update over the Head-connection. The return value of ('OK','U2') indicates that the request was completely sent. Since the chain proxy runs in Procedural Mode this implies that the chain proxy is ready for the next Update. The UpdateReadyHandler() will not be invoked. When the reply arrives the UpdateReplyHandler() is invoked for U2, with the result of the operation being "ok".

**6).** UpdateAsynch() is attempted while the Head-connection is down. The chain proxy is not able to send the request immediately and therefore returns ('INIT','U3') immediately. More interestingly, after this nothing happens. Especially there will be no timeouts. Recall that non-blocking operations do not block-on-send. Moreover, the ReplyTimeout is only activated *after* the request has been completely sent. This will not occur until the Head-connection is repaired in step 7). Before this happens the chain proxy will not be ready to accept new Updates. At this point a second attempt at UpdateAsynch() will simply yield (NOOP, None).

**7).** The chain proxy receives a notification from the chain that there is a new Head and that the chain length is reduced. This causes the ChainLengthHandler() to be invoked. After this, the chain proxy attempts to set up the new Head-connection. As soon as this succeeds the blocked Update from 6) will be sent. When it is sent, UpdateReadyHandler() is invoked to signal the readyness of the chain proxy to accept new Updates. Finally, the UpdateReply is received and the UpdateReplyHandler is invoked.

**10).** UpdateAsynch is invoked while the Tail-connection is down. The operation returns immediately after having sent the request successfully on the Head-connection. However, since the Tail-connection is down the UpdateReply times out. The UpdateTimeoutHandler() is invoked. Despite this the chain proxy will be ready for new Updates as long as the Head-connection is live.

**12).** This step is similar to step 7). An operational Tail-connection unblocks the Query from step 11), causing QueryReadyHandler() to be invoked and, eventually the QueryReply to arrive. Notice that the QueryReply reveals the effects of the Update that timed out in step 10). Thus, the Update was lost after being processed, not before.

## 11.5 Correctness of the Session-Protocol with Chain Failures

The following suite of experiments study the effects of failures in the session-start protocol and the session-stop protocol. The experiments are very similar to the ones described above, so we do not print the log for each experiment. Instead we simply describe the test and discuss the results.

**1) Session Start fails by not being able to connect to the Chain.**

There are many reasons why the initial connect may fail (e.g. host not found or connection refused). We have tested this failure by trying to start a session before the chain simulator has started. Then we start the chain simulator and try again.

| | **Blocking** | **Non-Blocking** |
|---|---|---|
| **Fail** | >>> **SessionStart**("headIP")<br>('NOOP', 'None') | >>><br>**SessionStartAsynch**("headIP")<br>('INIT', 'U1')<br>... SessionStartAbortHandler(NOOP) |
| **Success** | >>> **SessionStart**("headIP")<br>('OK', '') | >>><br>**SessionStartAsynch**("headIP")<br>('INIT', 'U1')<br>... SessionStartHandler() |

The synchronous operation will block until SendTimeout expires. If it does expire, the session-start protocol is rolled back, and the operation unblocks. NOOP is returned to signal that the chain proxy is in exactly the same state as before the operation.

The asynchronous operation does not return NOOP. This is because SessionStartAsynch() returns immediately, before it is known whether the operation will succeed or not. The SendTimeout is used to detect the failure of the connect. If the Register request is not successfully sent before SendTimeout expires, the session-start protocol will be rolled-back and SessionStartAbortHandler() will be invoked with NOOP as parameter. If the operation succeeds SessionStartHandler() is invoked.

**2) Session Start fails because the RegisterReply times out.**

After the Head-connection is set up and the RegisterRequest is completely written to the network, there are still a couple of things that might go wrong. The chain may simply drop the message, either before or after processing it. Or, the establishment of the Tail-connection may fail. In either case this looks the same for the chain proxy. The RegisterReply times out. We achieved this effect by having the chain simulator deliberately dropping messages.

| | **Blocking** | **Non-Blocking** |
|---|---|---|
| **Fail** | >>> **SessionStart**("headIP")<br>('TIMEOUT', 'None') | >>><br>**SessionStartAsynch**("headIP")<br>('INIT', 'U1')<br>...<br>SessionStartAbortHandler(TIMEOUT) |

The synchronous operation will block like in the previous experiment, but this time

TIMEOUT is returned instead of NOOP. TIMEOUT indicates that the RegisterRequest may have been processed by the chain. In contrast, NOOP indicates that the RegisterRequest was definitely not processed by the chain.

The asynchronous operation is basically identical to the previous example. This is a problem, because it may be important to be able to distinguish scenario 1) and scenario 2). This is the reason why SessionStartAbortHandler() is invoked with a parameter, NOOP or TIMEOUT. Again, TIMEOUT implies that the RegisterRequest was successfully sent, but that RegisterReply did not arrive in time.

So, the behaviour of the chain proxy is well defined with respect to lost Register Replies. Still, a lost Register Reply may seem to be a problematic situation. This is because the chain client does not know whether the first Register Request was processed by the chain, or not.

a. The RegisterRequest is dropped by the Head before being processed. As a result, the Head-connection will be established, but not the Tail-connection. The chain proxy and the chain will both agree that no client-session has been established.
b. The session-start protocol is run almost to completion, but in the final step the Tail just skips establishing the Tail-connection, for some reason. In this case the chain proxy will have to abort the session-start protocol. However, from the perspective of the chain, the client-session is successfully established. This is because the client is now added to the client-list replicated along the chain.

There is no way for the chain proxy to distinguish these two scenarios. Recall also that Query operations are not allowed until the session-start protocol has completed. This leaves the chain client with only one option, try again.

Now, since the chain client will have to restart the session-start protocol, what we need to ensure is that this has the same effect in the chain, independent of which scenario a) or b) caused the previous RegisterReply to time out. Consider what happens when the session-start is restarted after scenarios a) and b) respectively.

a. The Head-connection is already established, so the RegisterRequest is not delayed by this. The Register Request is then processed by all nodes in the chain, including the Tail.
b. The Head-connection is already established. The Register Request is then processed by all the nodes in the chain for the second time. However, since the Register operation is implemented as an idempotent operation, these are effectively NOOP's.

Now, note that after the Tail has processed the RegisterRequest, the two scenarios have become identical. The second attempt on session-start either transforms a) into b) or preserves b). This illustrates why it is crucial that the Register operation within the chain is implemented as an idempotent operation.

We have tested this by dropping a single RegisterRequest or RegisterReply. As long as the same thing does not happen again when the session-start protocol is restarted the protocol is able to complete sucessfully.

**3) Session Start fails because Head or Tail is down.**

We also need to discuss how Head and Tail failures affects the session-start protocol. To be precise, what happens if session-start is initiated at precisely the time when the Head or Tail is down, and the chain has not yet been able to reconfigure?

First, Head is down when session-start is initiated. This implies that the chain proxy will not succeed at establishing the Head-connection. This is simply an instance of scenario 1).

Second, Tail fails during the session-start protocol. If it takes a long time before the new Tail is appointed, the RegisterReply will most likely time out as discussed in scenario 2). However, if the chain reconfigures rather quickly the session-start protocol should be able to complete, despite this hickup. We have tested this and the log is presented below.

| | **python consoleClient.py** | **python consoleSimulator.py** |
|---|---|---|
| 1) | | >>> **TailDown**(2) |
| 2) | >>> **SessionStart**() | |
| | ... ChainLengthHandler(2)<br>('OK', '') | |

The idea is to take the Tail down just before the session-start protocol is initiated by the client. We do this manually by executing step 2) as soon as possible after step 1). This way, in our chain simulator, the RegisterReply will arrive at the Tail while the Tail is still down. In this case we simulate the behaviour of a failed Tail by *not* creating the Tail-connection, as we would otherwise do. Instead the RegisterReply is simply blocked until the Tail is recovered at some point later. By specifying that the Tail is to be down for 2 seconds only, the Tail is able to recover and send the RegisterReply, before the UpdateTimeout expires at the chain proxy.
The log information from the chain simulator confirmes that this is indeed what happens.

**4) Session Stop fails because the Head is down.**

Session Stop will not fail because Head is down. Rather, it will behave like any other Update operation. The Unregister Request will wait in the OutHeadPort until eventually a new Head-connection is supplied.

**5) Session Stop fails because the UnregisterReply times out.**

If the session-stop protocol fails because of message loss, chain proxy behaviour is very similar to the session-start protocol (see scenario 2). A synchronous invocation will block until there is a timeout. The asynchronous operation returns immediately confirming that the Unregister is successfully sent. However, the SessionStopAbortHandler() is eventually invoked with TIMEOUT as parameter.

| | **Blocking** | **Non-Blocking** |
|---|---|---|
| **Fail** | >>> **SessionStop**()<br>('TIMEOUT', 'None') | >>> **SessionStartAsynch**()<br>('OK', 'U1')<br>...<br>SessionStopAbortHandler(TIMEOUT) |

When the session-stop protocol is aborted like this, the chain proxy reverts to the state it had before the protocol was initiated. Thus, the chain proxy will continue to think that its client-session is valid.

Again, the chain proxy can not really tell whether the Unregister Request was processed by the chain, or not. So it could be that the chain has ended the client-session by removing the client from the client list. This means that Unregister too must be implemented as an idempotent operation within the chain. This will allow the chain proxy to repeat the operation until it succeeds. We have tested this and it works fine.

**6) Session Stop fails because the Tail is down.**

Tail fails during the session-stop protocol. If it takes a long time before the new Tail is appointed, the UnregisterReply will most likely time out as discussed in scenario 5). However, if the chain reconfigures rather quickly one would hope that the session-stop protocol should be able to complete.

However, there is a problem with this. Consider the following scenario. The Unregister Request has been processed by all nodes in the chain, except the Tail. When the Tail receives the Unregister Request it simply fails. Now, eventually its predecessor in the chain will be appointed as the new Tail. The new Tail then has to re-establish a tail-connection with all the chain clients using the client list. And here is the problem. Since the chain has already processed the Unregister operation, the client is no longer in the client-list. In other words, the chain has terminated the client-session and will thus never open a new Tail-connection to that client.

So, a Tail failure causes the session-stop protocol to time out UnregisterReply as in scenario 5). Even worse, repeated invocations of the session-stop protocol will after this never be able to succeed. We have tested this and this is exactly what happens.

There are a couple of different solutions to this problem.

- Chain clients could simply shut down if they experiencing repeated timeouts when trying to terminate a client-session. This would pose no problems to the chain.
- The chain Head and the Tail could close Head-connections and Tail-connections which are not associated with any clients present in the client-list. With both connections closed the chain client will eventually experience a Session timeout.
- The chain proxy could run the session-start protocol first followed by a session-stop protocol. This will succeed as long as the chain has a working Tail. However, it will require some changes to the chain proxy. At present the chain proxy can not initate the session-start protocol in this situation. This is because it still thinks it is in a session.

When it comes to choosing one of these solutions the two first seem like the better options. Especially we want to avoid the third option. Allowing applications to restart the session-protocol in mid session, possibly to some other chain, does not seem like a wise thing to do.

## 11.6 Correctness Summarised

We have tested all the central primitives of the chain API (SessionStart(), Query(), Update() and SessionStop()) with respect to Head-failure and Tail-failure. This also included testing both the synchronous and asynchronous variant of each primitive. In this process we have identified and fixed many smaller issues. At this point we are not aware of any bugs in our implementation.

This evaluation has also forced us to discuss and define what the correct output should be in a variety of situations. In that sense this evaluation exemplifies and complements our earlier presentation of the chain API, especially the section about return values for various operations. We are now quite confident that the central functionality of the chain proxy is correct. Still, we have yet not been able to stress test these corner cases.

## 11.7 Evaluating Performance

At this point it would be interesting to quantify central properties of chain behavior, such as throughput, buffering, request-reply latency, and the effects of failures and recovery. It would also be relatively easy to do so, considering that the chain API (implemented by the chain proxy) eases development of all sorts of test applications.

Still, here we are focusing mostly on the behavior of the chain proxy, as opposed to the chain itself. Especially, we want to expose the behavior the chain proxy, as it is experienced by the application using the proxy. We investigate three common scenarios, all involving the processing of a given batch of Queries and Updates.

1. **No Failures**. In the first experiment all Queries and Updates are processed uninterrupted. This exposes proxy/chain behavior under normal conditions.
2. **Head Failure**. In the second experiment a Head failure is introduced during the execution of the Query/Update batch. This exposes proxy/chain behavior in the event of Head failure.
3. **Tail Failure**. In the third experiment a Tail failure is introduced during the execution of the Query/Update batch. This exposes proxy/chain behavior in the event of Tail failure.

For all three experiments we discuss the throughput and the latency for both the Query-stream and the Update-stream. First however we will describe the experimental setup which is shared by all three experiments.

### 11.7.1 Experimental Setup

The experiment is straight forward. A test application uses the chain proxy to issue a batch of Queries and Updates. At some point the processing of this batch is interrupted by a failure. Two timestamps are taken for each operation. One just before the request is issued and one just after the reply is received (by the application). From this data we can calculate both latency and throughput. Latency is the difference between the two. Throughput is defined to be the rate at which replies are received. The following defines the experimental setup.

**Chain Client and Chain Simulator.** The experiment is hosted by a single chain client and a single chain simulator. The two processes are hosted by one machine at Norut and one at the University of Tromsø. The University offers GigaBit network while Norut is not likely to provide more than 100 MegaBit. In any case the experiment is modest in term of bandwidth consumption. The Ping RTT is about 0.7 ms (in both directions). On both machines the network interface is shared with other processes. However, the network consumption of these processes are not expected to affect our experiment. Both the chain client and the chain simulator are implemented as single-threaded programs, thus there will be no overhead due to thread synchronization or thread-context-switches.

- University : Dell Precision Workstation 380 with 2.8 GHz Pentium D dual core.
- Norut : HP Proliant DL380G5 with 2x Dual Xeon 5130 2.0GHz (4 cores). Virtual Server on VMWare ESX 3.0 set up with 1 virtual cpu.

**Request Frequency.** Updates and Queries are delivered to the chain proxy (by the test application) according to two given frequencies; the update-frequency (**UF**) and the query-frequency (**QF**). Both frequencies are specified as number of requests per second. In our experiment both **QF** and **UF** are set to **10**. If the application at some point can not send a new request (due to chain failure) it will not drop the request. Instead it will try again later, according to the same frequency.

**Request Volume**. The experiment defines a total volume of Update-requests and Query-requests. The test application will launch requests according to the given request frequency, until all requests have been issued. We call this the query-volume (**QV**) and the update-volume (**UV**). In our experiment both **QV** and **UV** are set to **100**. Thus, issuing the entire batch should take about 10 seconds, with no interruptions.

**Transport Costs.** In our experiment, request messages and reply messages are simply chunks of data, with no structure or meaning to them. There are two chunk sizes. Small chunks are 128 bytes and big chunks are 1024 bytes (equals 8 small chunks). Chunk sizes are mapped to request/reply types in the following way.

- **Update** : BigChunk -> **Update Reply** : Small Chunk.
- **Query** : SmallChunk -> **Query Reply**: Big Chunk.

This implies that a real network cost is associated with the processing of requests and replies.

**Processing Costs**. Processing of Queries and Updates in the chain simulator are logically NOOP's. So, to simulate processing costs the chain simulator delays requests (before processing them) according to two values. Queries are delayed according to a query-delay (**QD**). Updates are delayed according to update-delay (**UD**). UD is the aggregate delay of processing and transporting an Update across the entire chain. This has previously been referred to as *Update Propagation Delay*. UD is a function of chain length. In our experiment **QD** is **0.1** second and **UD** is set to **0.3*N** seconds, where N is the chain length.

**Failures**. We test the chain proxy with respect to two different failure types; Head-failure and Tail-failure. Both failure types are associated with a failure duration (**FD**). This means that the recovery of Head-connection or Tail-connection will not be initiated until after FD seconds have passed. Recall that FD includes both failure detection and chain recovery. In our experiment we have chosen **FD** to be **3** seconds.

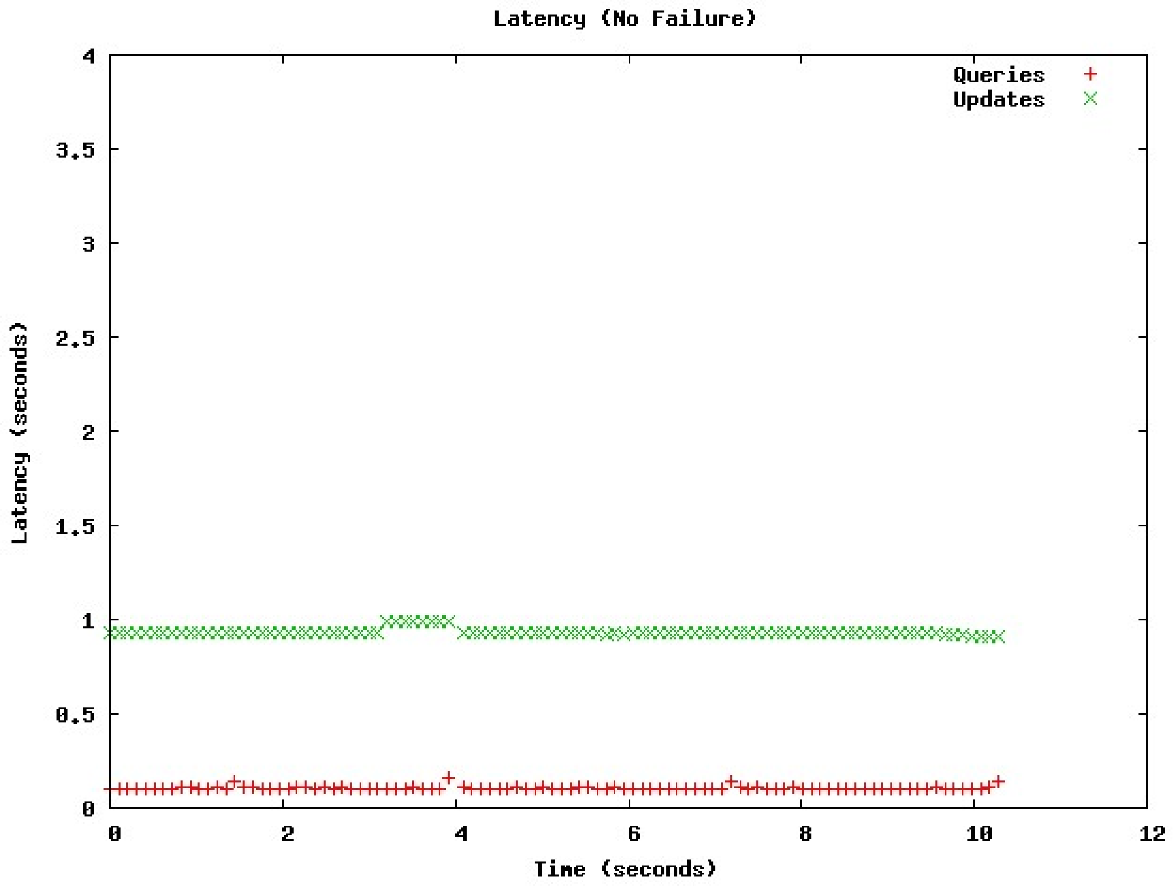
**Timeouts**. In order to measure the real effects of failures, timeouts are *turned off* in our experiment. This means that for every request sent by the application, the chain proxy will wait indefinitely for the corresponding reply. To match this expectation, the chain simulator is set up with an indefinite buffer space and will not drop any request/replies. However, there is one exception. Replies lost due to connection failures (i.e. messages on the wire as a connections is shut down) are reported to the application as timeouts. This only occurs when a *later* reply is received, indicating that the previous reply was lost.

**Message Loss**. Messages can be lost if they are in transit when the connection is taken down. However, the chain simulator takes down connections in a controlled manner, as opposed to a hard shutdown. This may result in a slightly lower probability of loosing messages. In addition, in a real chain Updates received by the Head, and not yet forwarded, will be lost too if the Head fails. No attempt is made to simulate this behavior. This could be done by say dropping newly received request on Head failure. Since we have not implemented such behavior our chain simulator will tend to drop less messages than a real chain.

### 11.7.2 Experiment 0 : No Failures

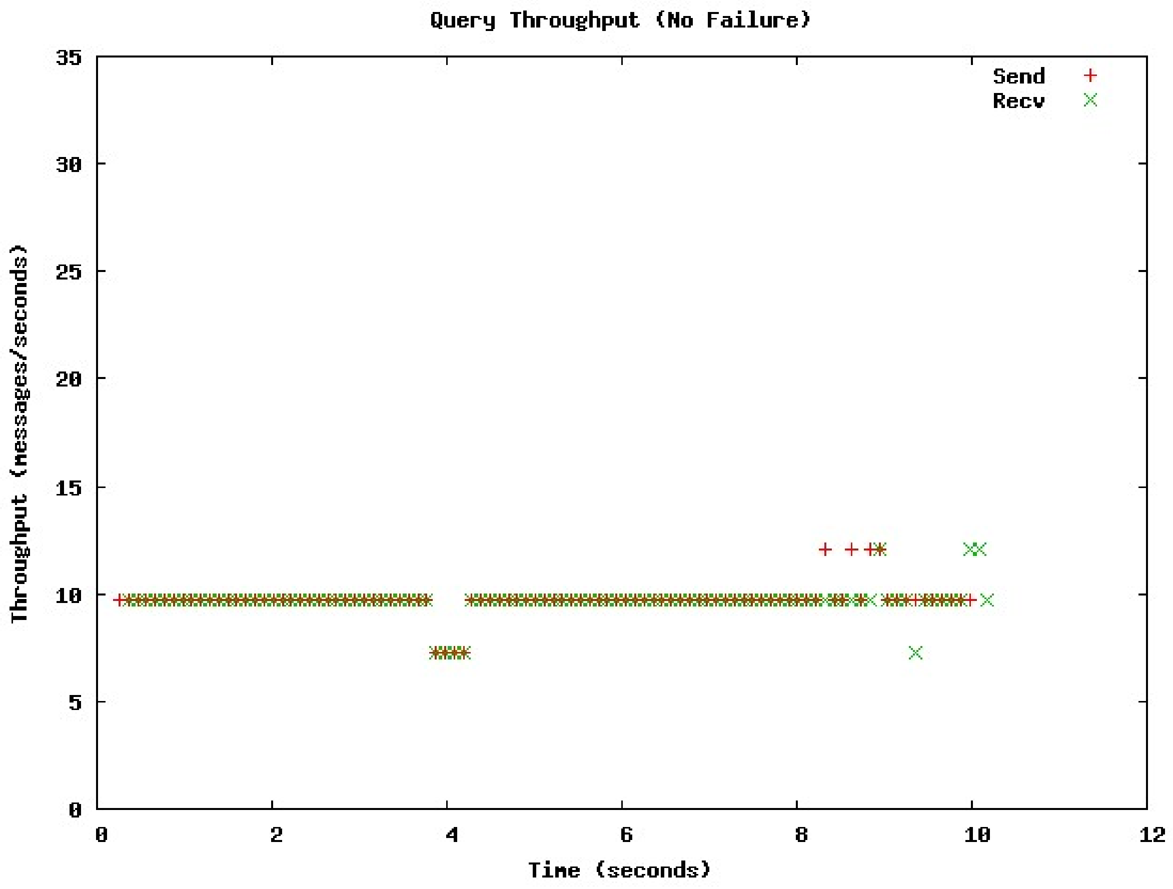
In this experiment a no failures occur during the experiment. We present default values for 1) Query/Update latency, 2) Query throughput and 3) Update throughput.

#### 1) Query/Update Latency (No Failure)



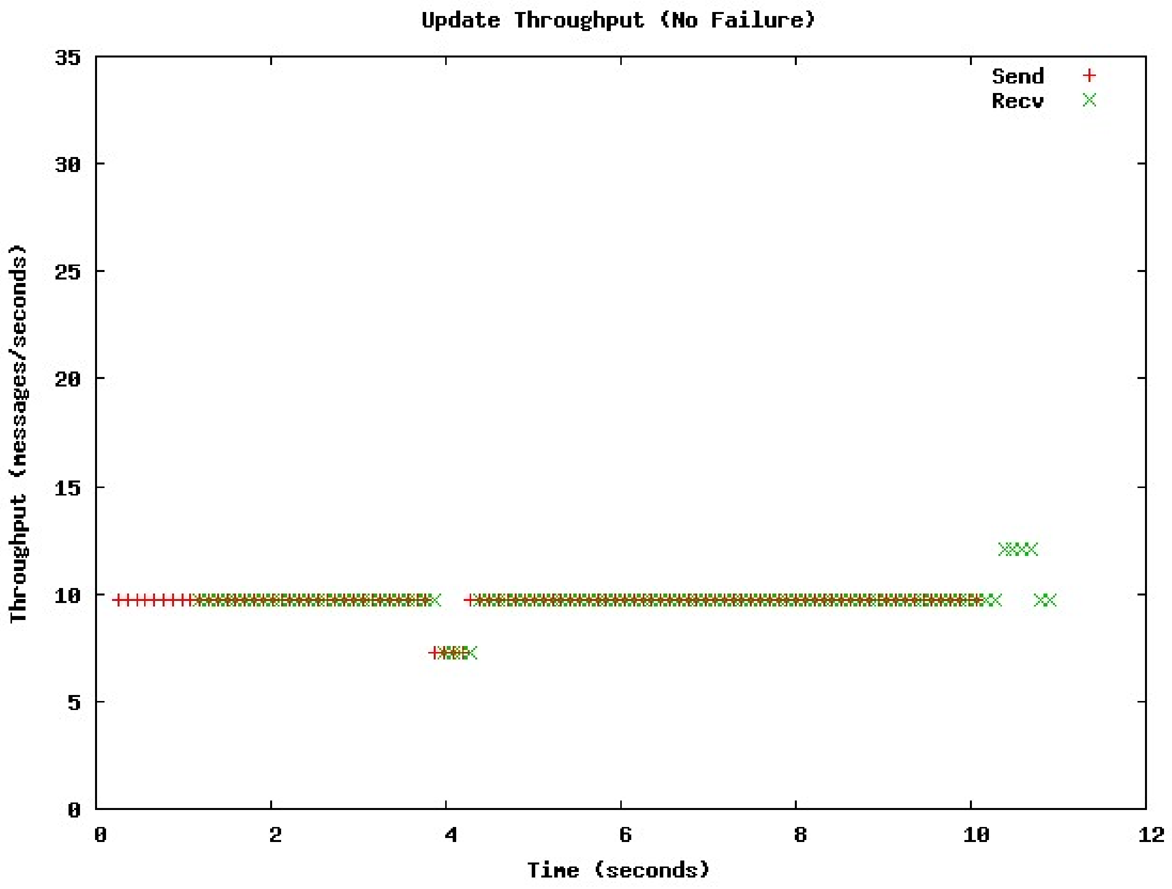
The first axis shows the time in seconds as the experiment progresses. The second axis shows request-reply latency in seconds. The entire experiment takes just over 10 seconds, as predicted by the request frequencies (QF/UF) and the request volume (QV/UV). The red dots show that Queries have a stable latency at just above 0.1 seconds. This is due mostly to the Query delay (QD=0.1s) enforced by the chain simulator. Similarly, the green dots show that Updates have a stable latency just above 0.9 seconds. This is due to the Update delay for a chain of length 3 (UD=3* 0.3s).

## 2) Query Throughput (No Failure)



From the same experiment we have also plotted the Query throughput. Throughput is measured in messages per second. The throughput is shown by two data sets. The red dots show the rate at which requests are sent. This should be equal to the Query frequency QF set for the experiment, which is 10. The green dots show the rate at which replies are received. As expected, the query throughput is close to 10. More importantly though, the receive rate follows the send rate quite nicely.

### 3) Update Throughput (No Failure)

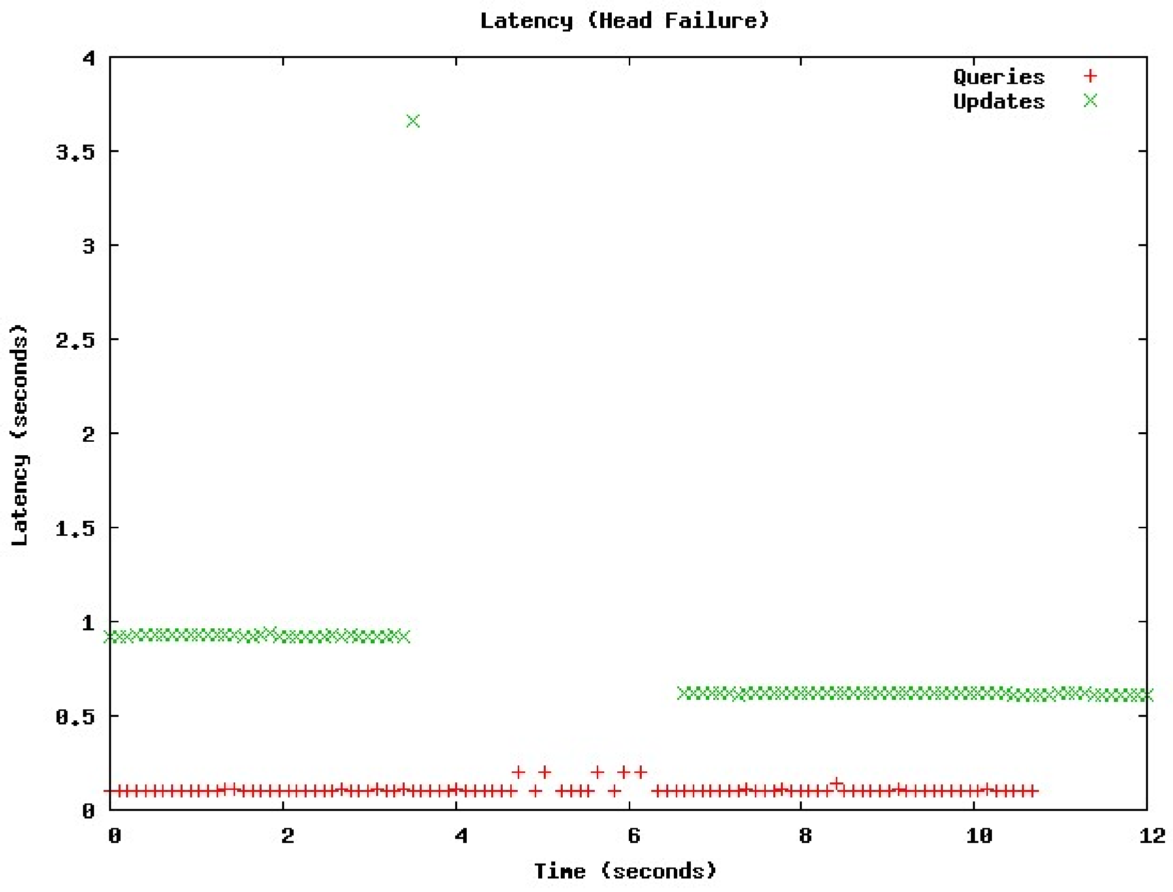


The update throughput also passes this test. The receive rate for replies lies close to 10 and it follows the send rate closely. Note also that the receive data series is delayed by about a second relative to the send data series. This corresponds to the update latency of just above 0.9 seconds. In this view it is rather surprising that the two data series both dip slightly at about 4 seconds into the experiment. However, we have not been able to reproduce this effect, so we can reasonably conclude that its cause must be an external event.

### 11.7.3 Experiment 1 : Head Failure

In this experiment a Head failure occurs during the experiment. We present implications for 1) Query/Update latency, 2) Query throughput and 3) Update throughput.

#### 1) Query/Update Latency (Head Failure)



Now we repeat the same experiment, except we introduce a Head failure after about 3 seconds. The Head connection is down for 3 seconds (FD) before the recovery protocol is initiated, so the connection should be recovered just before 7 seconds. No messages were lost in this experiment.

The first thing to notice in the plot is that the query latency (red dots) is mostly unaffected by this failure. This is according to expectation, since the processing of the Query stream is supposed to be independent of the Head connection.

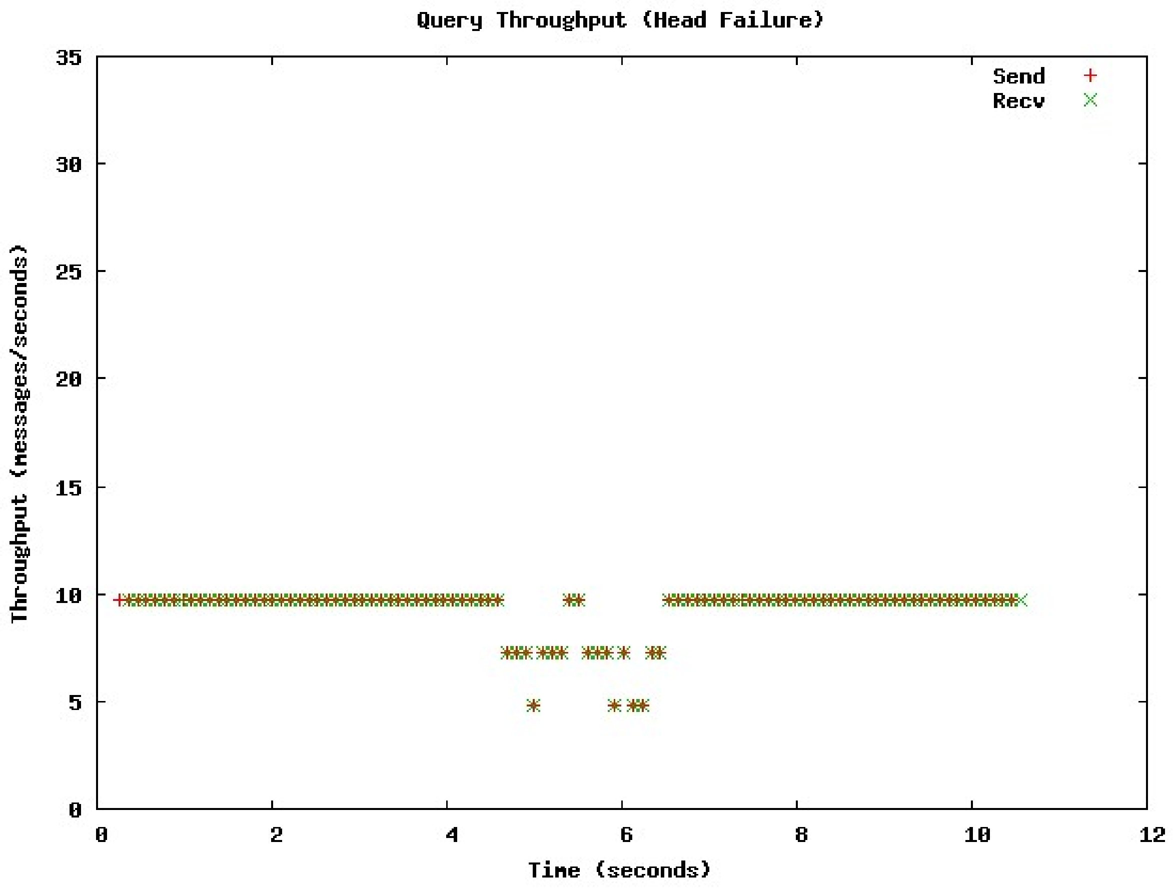
In contrast, the Update stream is hugely affected. For the entire duration of the failure, the Update stream has no progression at all. Note also that the increased update latency is not distributed among multiple update request. Instead, a single update request gets a very high latency of about 3.7 seconds. This is the request sent just after the failure occurred. It can not be successfully sent until the connection is repaired, and no other Update requests will be sent until the problem has been fixed. So, the rest of the Update requests experience normal latency. In fact, the latter part of the request batch experience a lower latency. This is because the failure causes the chain length to decrease. A chain of length 2 has an Update delay UD of 0,6 seconds, which can clearly be seen in the plot.

The value of the outlier latency (3.7 seconds) is interesting. It is the time from the application issues the first request (after a failure) until the corresponding reply is received. We can calculate an expected value for this latency. It would be the sum of the following intervals.

1. The time from the failure occurs until the application sends next update request. Since we send a new request every 0.1 second, this is somewhere between 0.0 and 0.1 seconds.
2. The duration of the failure, 3.0 seconds.
3. The time it takes to recover from the failure, i.e. send a notification and re-establish the head-connection, X seconds.
4. The time it takes from the delayed request is sent until the corresponding reply is received. This is the Update latency of a length 2 chain, about 0.6 seconds.

This gives us an expected time of about 3.7 + X seconds. Since the measured value is 3.7 we may be assured that the overhead of the recovery protocol (X seconds) is negligible compared to the duration of the failure (3.0 seconds).

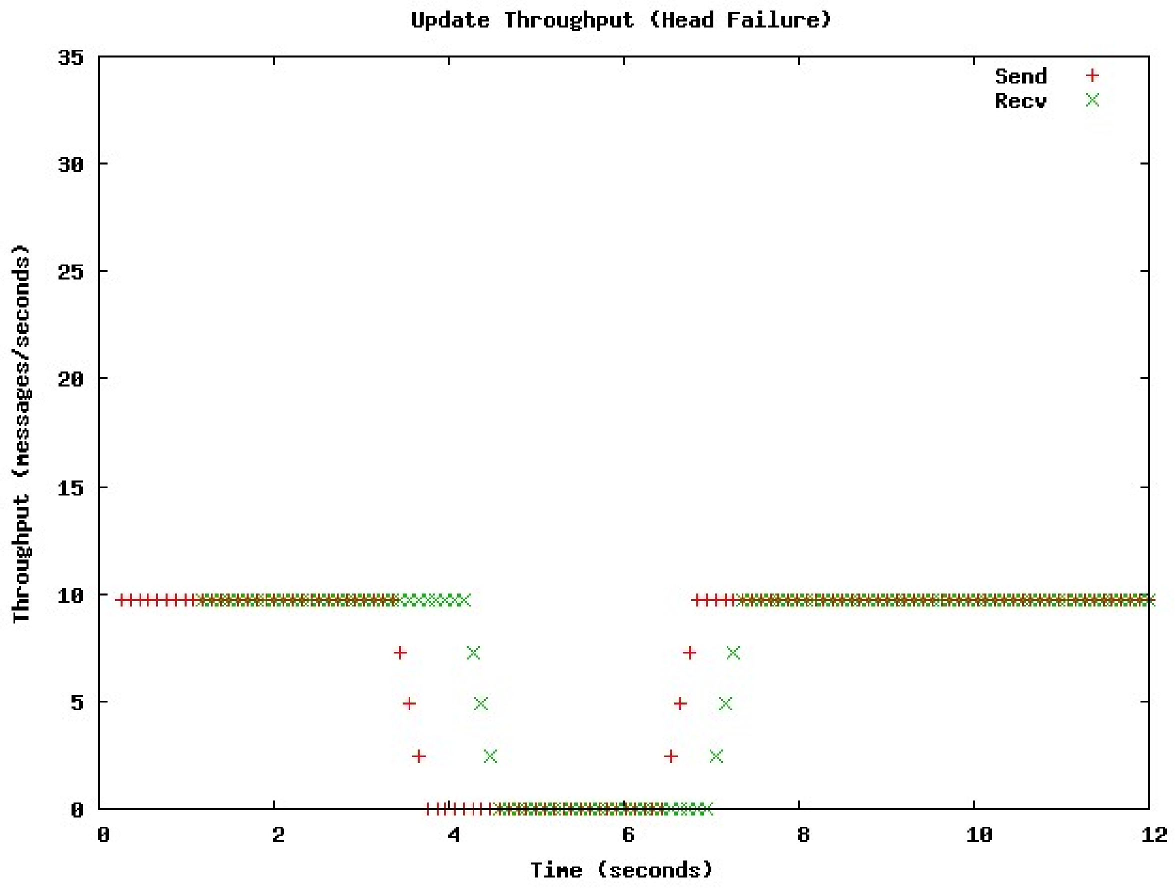
### 2) Query Throughput (Head Failure)



The Query throughput ideally is supposed to be unaffected by the Head failure, however this plot shows that the query throughput decreases slightly while the Head connection is down. The previous plot also showed a slight increase in latency in the same interval. Note however that the receive-rate follows the send-rate almost perfectly. This suggest that it is mostly the send-rate that is affected by the Head failure. We are able to reproduce this effect by repeating the experiment.

Overhead associated with the recovery protocol could be one possible factor in this. When the new Head-connection is to be re-established, this task may interfere with the sending of new Queries. However, this factor can not account for the entire dataset. The recovery protocol only runs to the very end of the outage, say at some point after 6 seconds. The dip in throughput happens before that. At the same time, the throughput dip does not occur immediately after the failure occurs. Only a second later do we see a reduction in send rate. At the moment we have not been able to identify the real cause for this effect. Still, we are able to reproduce this dip in query throughput, so it is not likely due to an external event.

### 3) Update Throughput (Head Failure)

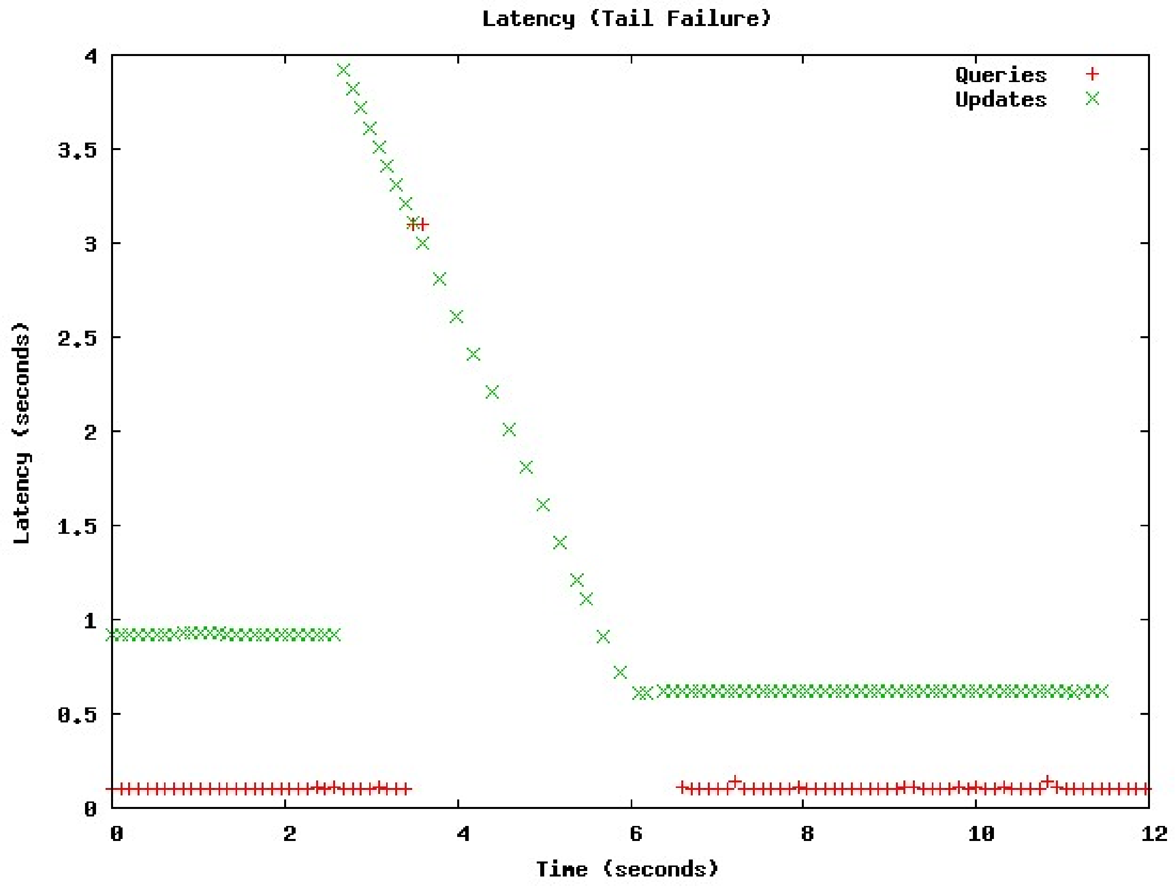


The plot of the Update rates is very much according to expectation. The send-rate drops to zero just after the failure occurred and jumps right up again as soon as the recovery has completed, after about 6.5 seconds. The receive-rate mirrors the send-rate perfectly, except it lags behind by about 0.9 seconds before the failure, and about 0.6 seconds after. This decrease in lag is due to the decrease in chain length (and thus to the decrease in Update propagation delay, UD).

### 11.7.4 Experiment 2 : Tail Failure

In this experiment a Tail failure occurs during the experiment. We present implications for 1) Query/Update latency, 2) Query throughput and 3) Update throughput.

#### 1) Query/Update Latency (Tail Failure)



In this last experiment we introduce a Tail failure about 3 seconds into the experiment. With respect to Queries the outcome is similar to the previous experiment with Head failure. The progress of the Query stream is brought to a complete halt until the recovery is complete. Note that two Query requests receive a very high latency, not only one as expected. This is due to message loss. The first Query was successfully sent before the Tail failure occurred, but was lost on the wire on its way to the chain simulator, or on its way back. In any case it will not be reported as lost until the next Query reply is received, which is delayed until the recovery is completed. As a consequence, the two requests share a very high latency. The value of this peak latency is also in agreement with our expectations. Repeating the above calculations yields an expected peak latency of 3.2 + X seconds. This too suggests a small overhead (X) related to recovery.

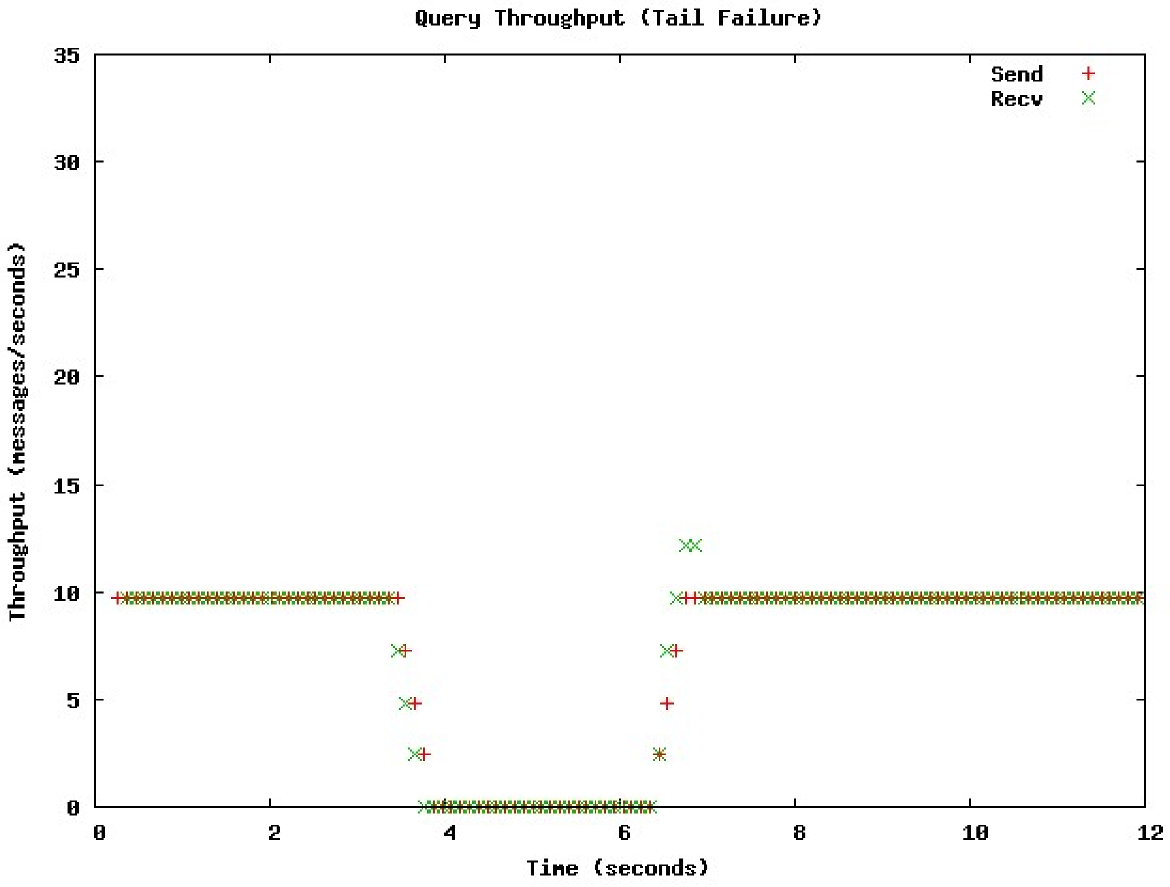
The Update stream behaves differently. This is because the send-rate of Update requests should not be very much affected by the Tail failure. The plot shows that the application continues to send Updates, but that the latency of each of those request is affected by the failed tail-connection. The experienced latency values fall on a straight line. This is only to be expected, since the latency of any given request is determined by the amount of time left before the tail-connection is recovered. Notice again how the latency stabilises at a lower level because of decreasing chain length. Notice also that the latency stabilises before the tail-connection is recovered. This is because an Update that is received 0.9 seconds before the tail-connection is restored will be delayed for that same amount of time, before it is processes. Thus, when the delay is completed the tail-connection is operational, thus normal latency is expected.

The maximum latency in this experiment is about 3.9 seconds. Here the expected value would be the sum of a similar sequence of steps as discussed above.

1. The Update request is sent to the chain simulator.
2. The Update request is delayed for 0.9 seconds.
3. The tail-connection goes down at most 0.1 seconds before the delay is terminated and stays down for 3.0 seconds.
4. The tail-connection is re-established, X seconds.
5. The Update reply is sent back to the proxy.

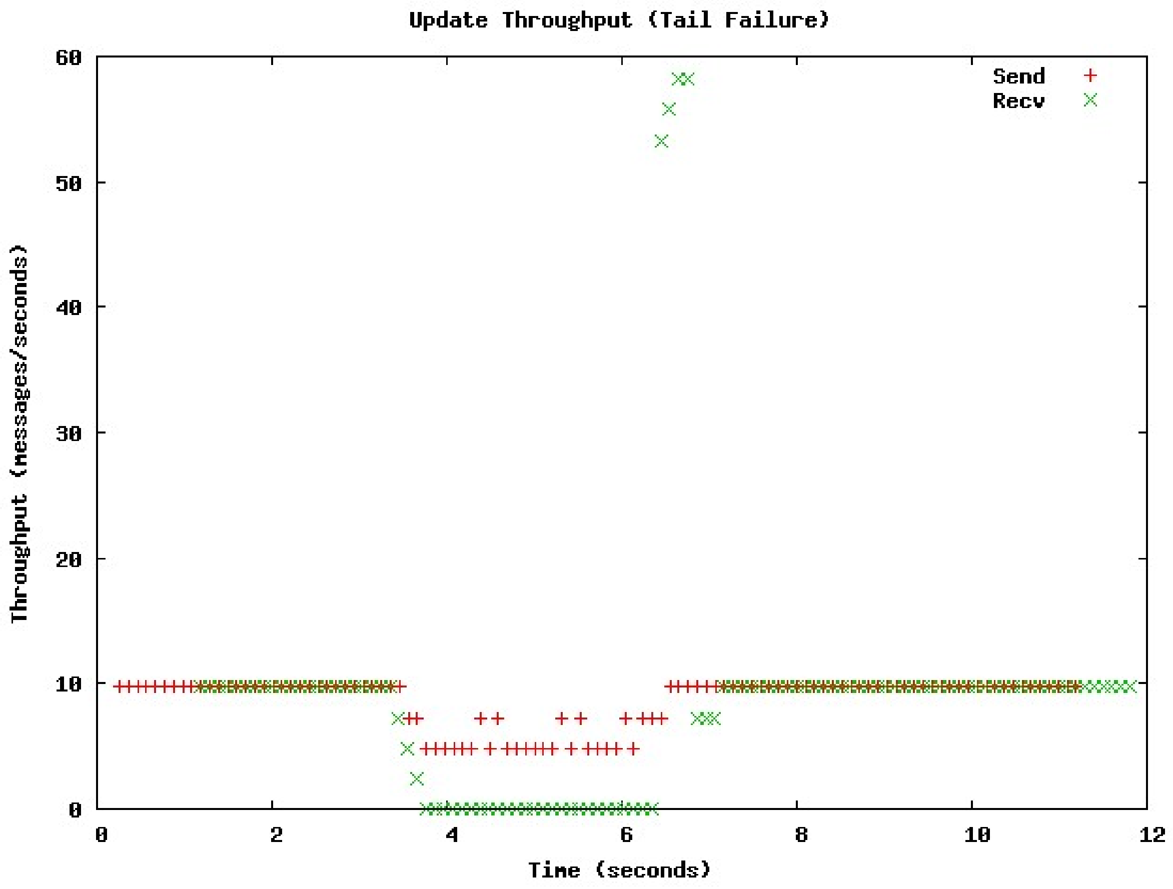
Step 1,2 and 5 together corresponds to the normal latency of a length 3 chain, just above 0.9 seconds. So, the expected maximum latency should be 0.9 + 3.0 + X seconds. Again, there is a very good correspondence between our expectations and the measured value. The overhead of re-establishing the tail-connection (X) is negligible compared with the duration of the failure (3.0s).

**2) Query Throughput (Tail Failure)**



The send-rate of Queries is directly affected by the tail-failure. It drops to zero as soon as the failure occurs and stays there until the failure is repaired. The receive-rate follows the send-rate quite smoothly, but deviates slightly in two cases. First, the receive-rate drops to zero at the same time as the send-rate. This is reasonable because the failure of the tail-connection affects both the Query-stream and the Reply-stream directly. There is also a small peak in the receive-rate right after the tail-connection is re-established. This is because Replies will be sent by the chain simulator for all pending Query requests. In addition, new Queries will produce new replies.

### 3) Update Throughput (Tail Failure)



Finally, the Update send-rate should be unaffected by the failure of the tail-connection. The plot shows that this is not exactly the case. The rate at which Update requests are sent seems to be slightly reduced for the entire duration of the failure. This is surprising. In fact, the non-arrival of Update replies during this period should make it easier for the proxy to keep the send-rate up to speed. This too is a repeatable effect. However, due to time constraints we must leave the exploration of this curious behaviour to future work.

The receive-rate of Updates behave as expected. It drops to zero as soon as the failure occurs and stays so until the problem is resolved. Then, as soon as the tail-connection is operational, all the Update replies that have been buffered by the chain simulator will be released in a burst. This accounts for the peek receive frequency of more than 50 replies per second. The amount of buffered replies should be about 30 at this time. Also, in this experiment we lost two Updates. This too contributes to a high peak receive-rate.

After the peak it dips slightly before it stabilises at the expected level. The small dip happens after all the buffered replies have been sent, but before new Update requests have completed their 0.6s delay.

## 11.8 Performance Summarised

In this evaluation we ran three experiments to expose the performance characteristics of the chain proxy. The experimental chain client issued a batch of Queries and Updates. This allowed us to profile the latency and throughput of all operations. In addition, we ran two  experiments to profile the behaviour of the chain proxy during chain failures. These are some of the important finds.

- All experiments
    - The parameters that defined the experiment set-up were visible in the results.
    - Stable latency and throughput values for both Queries and Updates.
    - Latency and throughput values reflected the realities of the network.
    - The receive rate of Replies matched the send rate of corresponding Requests.
- Head Failure
    - Query stream mostly unaffected by Head failure, but dips slightly.
    - Update stream comes to a complete halt.
    - Update latency is normal for all Updates, except one that will be delayed for the full duration of the outage.
    - Recovery of Head-connection is negligible compared to the duration of the outage.
- Tail failure
    - Query stream completely halted.
    - Update send rate mostly unaffected by Tail failure, but dips slightly.
    - All Updates delayed until recovery is completed. This defines Update latency.
    - Recovery of Tail-connection is negligible compared to the duration of the outage.
    - As soon as the Tail-connection is recovered, the chain proxy will receive a burst of delayed Queries and Updates.

# 12. Summary and Conclusion

## 12.1 Summary

In this thesis we have designed, implemented and evaluated a chain proxy for a chain replicated storage system. At the same time we revised the system architecture presented in [CR] by removing the dependency between chain clients and the chain master process.

Sections 2 and 3 provided the relevant background for this work. In section 2 we gave a brief introduction to replicated storage systems in general, and in section 3 we explained the chain replication protocol in detail. Section 4 presented the architecture of a chain replicated storage system. This enabled us to define the chain client, the chain proxy and all the relevant interfaces between the system components. Section 5 presented the session protocol that allowed us to remove the client-master dependency from the system architecture. In section 6 we presented the Message Port abstraction. This made it much easier to address the design of the chain proxy in section 7. Similarly, by discussing the implementation of Message Ports in section 8 we simplified the presentation of the chain proxy implementation in section 9. In section 10 we discussed the development of chain replicated applications, including a simple chain replicated file system, CRFS. Section 11 presented our evaluation of functional correctness and performance of the chain proxy.

## 12.2 Conclusion

We claim that the design and implementation of the chain proxy, as presented in this thesis, constitute an adequate solution to the problems put forward in the introduction. To back this claim we here repeat the problem definition and map it to the appropriate sections in the thesis.

> *1) Design a session-protocol that ensures the integrity of client-sessions even in the event of chain failures (i.e. Head-failure and Tail-failure). The protocol shall be executed by the chain clients and the chain exclusively (i.e. not require the master to participate).*

The session-protocol is defined and designed in section 5. The implementation of this protocol is presented in section 9 and evaluated in section 11. The correctness testing shows that the session-protocol implementation behaves correctly in a number of corner cases.

*2) Design and implement a chain proxy.*

   a. *The chain proxy shall encapsulate functionality common to all chain clients.*
   b. *The chain proxy shall provide programmers with a chain API in order to ease the development of chain-replicating applications. The chain API shall:*
      1. *provide flexibility and completeness.*
      2. *support a variety of applications types and programming models.*
   c. *The chain proxy shall implement the session-protocol defined in 1).*
   d. *The chain proxy shall strive to be as thin as possible. By this we mean:*
      1. *Minimise the overhead imposed by the proxy (latency and throughput).*
      2. *Expose the true behaviour of the chain, as opposed to masking it.*

Section 4 addresses 2a) by defining the interfaces that in turn defines the responsibilities of the chain proxy. The survey of the core logic of the chain proxy (section 7) is also done without reference to any specific application.

The two challenges of 2b) (2b.1 and 2b.2) concern the properties of the chain API. These challenges have motivated the discussion of the chain proxy design throughout section 7. The full chain API referenced to the end of that section is supposed to indicate both flexibility and completeness. The variety of supported application types (section 10) also backs this claim. The introduction of the concept of Port modes (section 6) was especially important with respect to the support of both event-based and procedural programming models.

Challenge 2c), the integration of the session-protocol in the chain proxy, is reflected by the chain API in section 7. Details concerning the implementation are additionally discussed in section 9.

Challenge 2d.1) is addressed in section 11. The performance evaluation indicated that the overhead introduced by our implementation is acceptable, with respect to both latency and throughput. However, we do not claim that the implementation is optimal. This is a best-effort prototype implementation. Still, the evaluation was able to confirm several desirable properties of our implementation. We showed that the Query stream and the Update stream were largely independent stream. In particular, the Query stream was unaffected by Head failure. Similarly, in the event of a Tail failure, the chain proxy could continue to send Updates.

Challenge 2d.2) is also addressed in section 11. The performance testing showed that the chain proxy is transient for applications, in the sense that it makes the underlying state of the chain (i.e. its failures) very much visible to applications. Head failure was shown to bring the outgoing Update stream to a complete halt. Similarly, Tail failure halted all streams except the outgoing Update stream.

Finally, we argue that this thesis sheds a little light upon one of the open questions related to chain replication. As argued in the introduction, the assumption that chain replication may implement quick recovery is a crucial part of the motivation for the protocol. This thesis addresses one part of this assumption, by indicating that recovery of client session can be negligible compared to the duration of the outage. Still, it remains to show that the recovery of the chain itself can be quick too.

## 12.3 Further Work

Much work remains before the presumed properties of the chain replication protocol can be verified by a real system implementation. In particular, a real implementation of the chain is needed. In this context it would be especially interesting if the master process could be made entirely obsolete.

It is our conjecture that this is indeed possible. The responsibilities of the master may be split in three parts. The master must 1) detect failures in the chain. Then, in the event of failures, the master must 2) ensure the integrity of the chain and 3) ensure the integrity of client sessions. These three tasks must all be implemented reliably within the chain, in order to remove the master entirely from the system architecture. In this thesis we were able to solve 3), so what remains as further work is 1) and 2). Note also that our chain proxy implementation will not be affected by such a modification to the system architecture.

In fact, the prototype of the chain protocol that we developed earlier did not require a master process, since it included protocols that solved all challenges 1), 2) and 3) completely within the chain. Unfortunately, results from this experience has not yet been published. Also, that first implementation does not fully comply with the client-chain interface that is required by this chain proxy implementation. So, a reasonable next step would be to re-implement the chain prototype, and test both chain and chain proxy together. In addition, an application needs to be implemented that is able to fully exploit the pipelining properties of the chain replication protocol.

## References

- [CR] R.v. Renesse and F.B. Schneider, "Chain Replication for Supporting High Throughput and Availability", *OSDI Operating Systems Design & Implementation*, pages 91–104, 2004.
- [MON] C.A.R. Hoare, "Monitors: and operating system structuring concept" Communications of the ACM, 10(17):549-557, 1974.
- [RPC] A.D. Birrell and B. Nelson, "Implementing remote procedure calls", *ACM Transactions on Computer Systems*, 1(2):39-59, 1984.
- [GFS] S. Ghermawat, H. Gobioff, and S. Leung, "The Google File System". In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, October 2003.
- [SMA] F.B. Schneider. "Implementing fault-tolerant services using the state machine approach: A tutorial". *ACM Computing Surveys*, 22(4):299-319, December 1990.
- [PBA] N.Budhiraja, K. Marzullo, F.B. Schneider and S. Toueg, "The primary-backup approach", *Distributed Systems*, pages 199-216, ACM Press, 1993.
- [PAST] A. Rowston and P. Drushel, "Storage management and caching in PAST, a large scale, persistent peer-to-peer storage utility", In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Banff, Canada, October 2001.
- [CFS] F. Dabek, M.F. Kaashoek, D. Karger, R. Morris and I. Stoica, "Wide-area cooperative storage with CFS", In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Banff, Canada, October 2001.
- [OCN] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao."OceanStore: An Architecture for Global-Scale Persistent Storage", In *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, November 2000.
- [RAID] D.A. Patterson, G. Gibson and R.H. Katz, "A case for redundant arrays of inexpensive disks (RAID)", In *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 109-116, Chicago, Illinois, 1988.
- [VTX] Å. Kvalnes, D. Johansen, A. Arnesen and R.v. Renesse, "Vortex: an event-driven multiprocessor operating system supporting performance isolation", Technical Report,  University of Tromsø, 2003-45.
- [HDP] Hadoop Distributed File System, http://hadoop.apache.org/hdfs/
- [FUS] FUSE Filesystem in Userspace, http://fuse.sourceforge.net/
- [WAIF] WAIF, Wide Area Information Filtering, http://www.waif.cs.uit.no/

## Appendix: Code Overview

The attached code is organised in a folder hierarchy as follows.

- **Master**/
    - **apps**/
        - EventSingleAsynch.py
        - ProcMultiSynch.py
        - ProcMultiAsynch.py
        - ProcSingleAsynch.py
    - **chain**/
        - __init__.py
        - chainSimulator.py
        - common.py
        - consoleSimulator.py
    - **crfs**/
        - common.py
        - crfsClient.py
        - crfsChainSimulator.py
        - fusefs.py
    - **ports**/
        - __init__.py
        - tcpPorts.py
        - port.py
        - taskScheduler.py
        - udpObjects.py
        - tcpObjects.py
        - udpPorts.py
    - **proxy**/
        - __init__.py
        - chainProxy.py
        - consoleClient.py
        - misc.py
    - **exp**/
        - baseExpClient.py
        - expConsoleChainSimulator.py
        - frequencyClient.py
        - fullThrottleClient.py
        - common.py
        - logParser.py
        - **exp0**/
        - **exp1**/
        - **exp2**/
    - **doc**/
        - Master.pdf

**apps**/ includes some example programs. (~ 500 lines of python code).

**chain**/ contains the implementation of the chain simulator and the console interface. It also defines some common resources used by both the chain and the chain proxy. (~ 800 lines of python code)

**crfs**/ includes all code needed to run the fuse-based Chain-replication file system (CRFS). (~ 500 lines of python code)

**ports**/ contains the implementation of the message port abstraction, used by both chain proxy and the chain simulator. (~ 1700 lines of python code)

**proxy**/ includes the implementation of the chain proxy along with the console interface. (~ 1400 lines of python code)

**exp**/ contains scripts used to run experiments and analyse the results.
(~ 800 lines of python code)

**doc**/ includes the pdf of this thesis.

All in all, this project includes ~ 5000 lines of python code. The code is not sparsely commented though. In order to run much of this code the root directory (Master/) needs to be included of the PYTHONPATH. Some of the code in crfs/ additionally requires the Fuse-extension to be installed in the operating system and fuse-python-bindings to be available. These are the main executable programs.

- *python chain/chainSimulator.py* starts the chain simulator.
- *python chain/consoleSimulator.py* starts the chain simulator with a python interpreter console.
- *python proxy/consoleClient.py* starts the chain proxy with a python interpreter console.
- *python fuse/crfsChainSimulator.py* starts the chain simulator with the storage object required by the crfs implementation. This program requires a directory /tmp/fs to be available.
- *python fuse/fusefs.py /mountpoint* starts a user-level filesystem associated with the given mount point. (*fusermount -u /mountpoint* terminates the filesystem)

In addition all programs in the apps/ and exp/ directories may be executed.