Imperial College London

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Enhancing the Performance of Twitter Storm with In-Network Processing

Bachelor's Thesis

Clemens Lutz
ETH Zurich

Supervisors:
Dr. Peter Pietzuch
Dr. Paolo Costa
Large-Scale Distributed Systems Group
Imperial College London

Prof. Roger Wattenhofer
Distributed Computing Group
ETH Zurich

September 8, 2012

Distributed computation frameworks use large amounts of bandwidth to propagate input and result data between nodes. As the amount of nodes and the amount of data being processed on them grows, high-traffic routes on the network become a bottleneck. Cloud services offer their tenants no control over the network infrastructure, potentially leading to sub-optimal resource utilization. Current approaches try to overcome these shortcomings by using faster networking technologies or alternative network topologies.

In this paper, we explore the approach of complementing a generic switch with a general-purpose computer to form an in-network processing element and providing software-level support for controlling the locality of computations in a distributed computation framework. As such we make the network infrastructure more transparent to tenants of our computation framework, allowing them to implement application-specific data routing and processing. We call this model *Network as a Service (NaaS)*. Previous data center simulations of NaaS have shown promising results.

We present *NaaStorm*, a concrete implementation of a distributed computation framework modeled on the Network-as-a-Service concept. We show that in-network processing yields substantial speedups compared to the standard model.

In the process of implementing our worker module we have decided to forgo Twitter Storm integration and instead create a stand-alone distributed processing framework. We demonstrate our work in multiple benchmark applications.

Acknowledgements

I would like to thank Peter Pietzuch, Paolo Costa, Thomas Haddow and Matteo Migliavacca for their continuous advice, feedback and patience throughout the project. Without them the project would never have lifted off the ground, not to mention come to a conclusion.

Roger Wattenhofer, Bernd Gärtner, Herbert Wiklicky and Barbara Schori enabled me to come to London and write my thesis abroad. It's been an amazing time and has shaped me both intellectually and as a person. To them I would like to express my gratitude and encourage them to continue enabling other students to do likewise.

Most importantly I would like to thank my family, for they have always stood behind me to provide support and motivation throughout my life as a pupil and student.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Idea and Approach . . . 1
1.3 Contributions . . . 2
1.4 Report Structure . . . 3

2 Background **5**
2.1 Message-Oriented Middleware . . . 5
2.1.1 Overview . . . 5
2.1.2 ZeroMQ . . . 5
2.2 Serialization . . . 8
2.2.1 Overview . . . 8
2.2.2 Kryo . . . 9
2.3 Applications for NaaS . . . 9
2.3.1 MapReduce . . . 9
2.3.2 Twitter Storm . . . 10

3 Design **15**
3.1 Data Center Placement . . . 15
3.2 Storm Cluster Placement . . . 16
3.3 Storm Integration . . . 17
3.3.1 Worker Start-up and Heartbeats . . . 17
3.3.2 Communication with Storm Cluster . . . 18
3.4 NaaStorm Worker Architecture . . . 19
3.4.1 Task Framework . . . 19
3.4.2 Inter-task Communication . . . 20

4 Implementation **21**
4.1 Rationale . . . 21
4.2 Initializer . . . 21
4.3 Command-line Parser . . . 22
4.4 Communication Framework . . . 22
4.5 KryoCpp . . . 24
4.5.1 Serialization of Important Data Types . . . 24
4.5.2 Interface . . . 25
4.5.3 Internals . . . 25
4.6 Task . . . 26
4.6.1 Interface . . . 26
4.6.2 Internals . . . 28
4.6.3 Assembling the Pieces: An Example . . . 29

5 Evaluation **31**
5.1 Performance . . . 31
5.1.1 KryoCpp . . . 31
5.1.2 ZeroMQ . . . 33
5.1.3 NaaStorm . . . 34
5.2 Discussion . . . 37
5.2.1 Problems Encountered . . . 37

5.2.2 Limitations 38

6 Conclusion **41**

6.1 Future Work 41

Chapter 1

Introduction

1.1 Motivation

The ascent of the Internet has enabled people to share information in a scale previously unknown. Every day millions go online to communicate via services such as E-mail, instant messaging and social networking sites. In 2007 [1] estimated that 6 million users posted 65 million tweets on Twitter per day, predicting a doubling of posts each month. More recent numbers suggest 100 million active users[2] and 200 million tweets per day[3].

Researchers and enterprises have harnessed this massive amount of content generated. For instance, [4] explores how sentiment analysis can predict stock market movement. Tweets stating explicitly the mood the author is in are extracted by searching for terms like "I'm feeling" and "makes me". These are passed to tools which output moods such as calm, alert, sure, vital, kind and happy. Using machine learning techniques the authors establish a link between e.g. "calm" and the Dow Jones Industrial Average.

Sakaki et al [5] use social networks to detect earthquakes in real-time. Each user is considered as a sensor, each tweet as sensor information. Keywords such as "earthquake" and "shaking" return possibly relevant posts which are classified using machine learning techniques. An approximate location estimation is made by reading users' GPS coordinates. The authors show that in most cases notification E-mails are sent within a minute, with a detection rate of 96% for JMA intensity scale 3 earthquakes. In comparison, official channels have a response time of 6 minutes.

Processing massive amounts of data in real-time can only be achieved by distributing the workload across many computers. While software frameworks such as Apache Hadoop, Google's MapReduce and Twitter Storm exist for this purpose, studies have shown [6, 7] that their scalability is limited by the throughput of core-level switches. Traditional data center networks are strictly hierarchical and are divided into core, aggregation and access layers [8]. In order to maintain that structure the common approach taken is to install faster hardware to alleviate the bottlenecks. Scaling up is expensive and is limited by the technology available.

An alternative is the use of non-standard network topologies [9, 10, 11] or multi-path networks[12, 13]. However, these come at the cost of custom routing protocols, management and maintenance overhead and may exist only for specialized systems.

SideCar[14] presents a conventional network topology complemented by off-the-shelf servers connected to each switch via redirection mechanisms. While the paper presents an approach to pervasive network instrumentation and programmability, their extended network topology can be re-purposed for a novel approach to distributed computation frameworks.

1.2 Idea and Approach

In their paper[15], Costa et al argue that in-network processing can benefit tenants of cloud infrastructure and platform services by giving them application-level control over network forwarding decisions. Traditional network infrastructure separates computation in the end hosts from end-to-end routing in the network. They argue that the separation negatively impacts both performance and flexibility. In their model, applications implement custom data aggregation, stream processing, caching and redundancy elimination protocols assisted by in-network processing elements. These are either connected via high-bandwidth links to switches or integrated into switch hardware. They coin this network architecture and programming model *Network-as-a-Service (NaaS)*. Simulations show promising results, with greatly increased application performance and reduced overall data center traffic.

In this paper we aim to demonstrate the practical viability of NaaS by applying the model to a modern distributed computation framework. We aim to achieve a significant speedup of reference applications on our modified framework when compared to the plain vanilla installation, both through the efficient bandwidth utilization of in-network processing and through our machine-native implementation. Our goal for raw throughput is to achieve at least 1 Gib/s, ideally 10 Gib/s. Like in all real-world scenarios there are applications which experience little to no speedup. As such we will identify both positive and negative examples of programs suitable to the NaaS model.

We build our system, called *NaaStorm*, to integrate with the distributed real-time computation framework Twitter Storm[16]. Building upon an existing framework benefits us by providing:

- a tested and well-performing base, eliminating many frequent pitfalls when constructing a system from scratch,
- a convenient and fully-featured interface, which is usually not a consideration for research platforms,
- a proven track-record with companies and an existing application base, substantiating our aim of showing real-world applicability and
- a user base with mailing list and wiki, which can be helpful for solving common problems.

In short, building upon an existing framework allows us to focus on the relevant aspects of implementing a scalable in-network processing system.

Twitter Storm was selected as our basis for the following reasons:

- Social network
  Twitter[1] is one of the world's largest social networks. Many previous research projects in a variety of fields have resorted to feeds of user tweets for relevant input. As such Twitter provides us with wide range of interesting usage scenarios and data streams with Storm acting as a catalyzer.
- Active project
  Twitter Storm is a young project which is actively developed. A modern and manageable code base is also beneficial as it largely liberates us from dated code and crude APIs.

1.3 Contributions

This project has made the following contributions:

- Network-as-a-Service concept demonstration (chapters 3, 4, 5.1.3)
  We showed by way of an implementation how a Network-as-a-Service architecture can be designed and constructed in practice. Its gains and drawbacks have been evaluated with several demo application benchmarks.
- Design of external component integration into Twitter Storm (sections 3.3, 4.2)
  We showed that it is feasible to integrate components of an external project into the Storm framework, independent of the component's implementation language and internal architecture, by way of a step-by-step design and a prototype wrapper layer.
- Kryo serialization format portability (section 4.5)
  We designed a Kryo-compatible serializer in a non-JVM language and showed that it can be used for a concrete project.
- Serializer evaluation (section 5.1.1)
  We evaluated the performance of the Kryo and KryoCpp serializers and showed their characteristics when encoding a selection of data types.
- ZeroMQ evaluation (section 5.1.2)
  We evaluated the performance of the ZeroMQ message-oriented middle-ware and showed that it's use has minimal performance impact compared to TCP/IP.

[1]http://twitter.com

1.4 Report Structure

First we give an introduction to the relevant concepts and projects behind our work in the Background (chapter 2). We give an overview of Twitter Storm and its foundations, ZeroMQ and Kryo.

In the Design chapter (3) we describe our design of NaaStorm and its logical and physical placement in the cluster. We try to give insight into our decisions and discuss the alternatives.

Then we immerse into the implementation details (chapter 4) of our project. Here we explain the inner workings of our system and how we implemented a Kryo-compatible serializer. Again we discuss alternatives and justify our decisions.

Chapter 5 provides graphs and numbers of the performance results achieved. We analyze them in detail and try to give a sensible interpretation. We discuss our project and its limitations. We provide insight into what challenges and problems we had during the design and implementation phases.

Finally, we conclude (chapter 6) our project, reflecting what we have learned and decisions we would make differently the second time. We summarize the goals we have achieved and discuss those which have eluded us. We give an overview of how our project could be extended and future work.

Chapter 2

Background

In this chapter we give an overview over related work and the background information necessary to understand our project's design and design considerations.

2.1 Message-Oriented Middleware

2.1.1 Overview

Traditionally application-layer communication is done either directly with low-level socket programming, as with the TCP and UDP protocols, or with remote procedure calls.

Low-level sockets establish a uni- or bidirectional point-to-point connection between two end points. Transmissions are efficient and fast. However, implementations are specific to the programming language and operating system, limiting platform portability. Their use is often cumbersome, leaving the developer to handle multiple targets and sources, communication patterns and failures.

RPC enables locally called procedures to transparently executed on a remote host. While this facilitates a number of applications, it is inflexible and reliability semantics vary between implementations. Furthermore, RPC implementations are often bound to a certain programming environment.

[17] defines message-oriented middleware as "any middleware infrastructure that provides messaging capabilities". Aiming for flexibility, message-oriented middleware is an abstraction layer which generally provides multiple interaction models, reliability, availability and security, with both synchronous and asynchronous semantics[18, 19].

Several common interaction models are point-to-point, request-reply, publish-subscribe and push-pull. Request-reply is found in in many internet applications, e.g. file and HTTP client to server pattern, publish-subscribe is found in event processing systems. Push-pull is a basic pattern for unidirectional communication.

There are many proprietary and open-source message-oriented middlewares on offer. However, we only describe ZeroMQ in greater detail as it forms the basis of Twitter Storm's communication layer.

2.1.2 ZeroMQ

Overview ZeroMQ[20] is a fast, modern networking protocol implemented in C++. It abstracts low-level communication to provide a more convenient and consistent API. Multiple common patterns are implemented in ZMQ sockets for both local and remote connections. Unless otherwise specified, transmissions are sequenced, reliable and connection-oriented. Furthermore, by default transmissions will block until both source and destination are ready. Multiple senders and receivers on a single connection are supported. A message can be attributed to a node as each node has an identifier.

Socket Types A description of the most important ZeroMQ socket types for this project:

Push and Pull *ZMQ_PUSH* and *ZMQ_PULL* represent a pipeline communication pattern.

A source node pushes messages to one or multiple destination nodes. Each destination node can pull messages from one or multiple source nodes. Communication is unidirectional, meaning sending is on pull sockets and receiving on push sockets is undefined. When multiple senders are active received messages are fair-queued. When sending to multiple receivers messages are round-robin-ed. If a predefined sending window is full or no receiver is ready the sending socket blocks.

An example use case of push and pull is Map-Reduce. A mapping process distributes working sets among multiple worker nodes. Their output is sent to a reduction process. ZeroMQ takes over the responsibility of scheduling as messages are queued when sending and receiving.

Pair *ZMQ_PAIR* represents a one-to-one connection. Communication is bidirectional, meaning both nodes at either end of the connection can send and receive. However, because a single ZMQ socket cannot be used by multiple threads at the same time, listening for incoming messages and sending must be alternated. Take note that a pair connection is only defined for local, inter-thread communication.

Dealer *ZMQ_DEALER* represents a one-to-many connection. A dealer distributes messages to multiple receiving nodes in a round-robin fashion. Connections are bidirectional and incoming messages are fair-queued among nodes. If the sending window is full or no receiver is ready the socket blocks on send.

An example use case is a message broker. On one end a specified socket receives messages. These are passed to a dealer socket, which distributes them among several worker nodes or threads. The workers send their output to the dealer, which passes it on to the outside socket.

Router *ZMQ_ROUTER* represents a many-to-one connection. A router receives messages from multiple sending nodes in a fair-queued fashion. Connections are bidirectional, outgoing messages are sent to the node specified in the identifier field. If the receiving node is not ready the message is dropped.

Router sockets allow raw access to messages node identifier. Each message consists of at least two parts. The first part is the node identifier, specifying the origin node if the message was received or the destination node if the message is to be sent. ZMQ identifiers are between 1 and 255 bytes long, where the first bit must not be 0. The second part and all further parts are payload data.

An example use case is a server application. A router socket listens for incoming requests. Each request is processed and the result returned to the corresponding client. Take note that requests can be processed asynchronously as each client can be linked to its request by the ZMQ identifier.

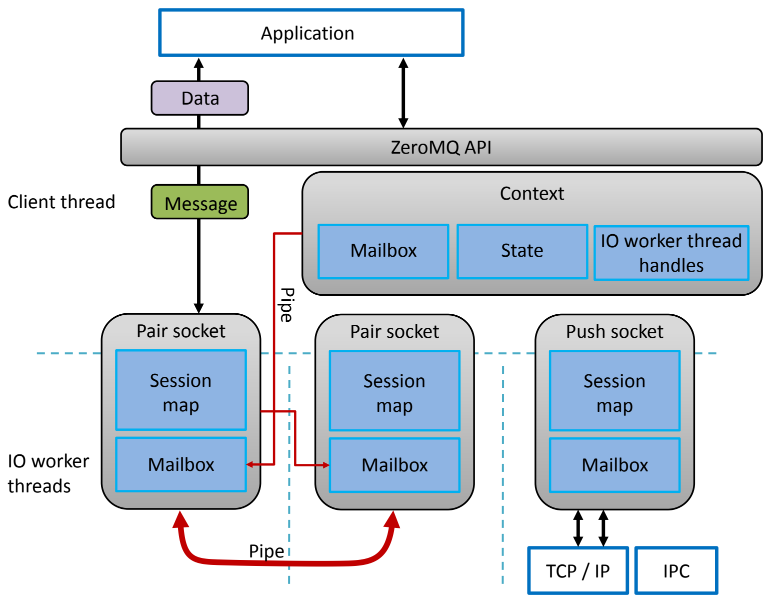


Figure 2.1: ZeroMQ Overview

Design and Implementation

Structure ZeroMQ is a large project which involves much inter-thread communication. Figure 2.1 gives a high-level overview of the structure and the allocations of components to threads. API calls and all subsequent operations up to the queuing of messages and notifications of worker threads are performed in the client thread. IO worker threads then handle the network, inter-process or inter-thread

communication.
Noteworthy components are:

- User-facing API: Consists of the context, socket and message interfaces. A context has one or more sockets, contains a system notification mailbox object, IO worker threads and holds state information shared between its sockets. The socket class is a proxy[21] to the socket base class, itself sub-typed by the socket types described in 2.1.2. Each socket has a mailbox, session map with open connections and corresponding mutex. A message is simply a struct. It notably contains a pointer to the payload with function pointer to a destructor and a reference counter which is incremented on message copying.

  ZeroMQ provides both C and C++ APIs. The C++ API is a proxy to the C API. The implementation of the C interface in turn references components in the library, which is exclusively written in C++.

- Communication framework: The foundation of the communication framework is the pipe. Pipes are the means of transportation used between sockets, IO threads and user threads. For example, the ZeroMQ inproc protocol utilizes pipes for inter-thread communication in a process.

  A pipe is made up of a reader, writer and queue. Reader and writer classes represent facades[21] to the queue. Two main goals are achieved by abstracting:

  - Client communication: To handle concurrency efficiently the writer explicitly makes previously written elements accessible to the reader by executing a flush command. Flushing updates the only shared variable, a sentinel which stores the location of the last cleared element. The reader may then access all elements up to that point and stores the last known sentinel locally. The cached sentinel is updated upon reading past it's location. Furthermore, when the queue has reached a defined maximum length, the reader will signal the writer on detaching elements.
  - Send rate limitation: The *High Water Mark* is enforced in the writer by limiting the queue length.

  The queue is modeled on a doubly linked-list with arrays as elements. Head and tail array indexes and pointers to head and tail array chunks are stored. The data structure itself does not provide guarantees on concurrent access, thus a wrapper implements lock-less concurrency. On flush and sentinel prefetching the sentinel is updated via a CMPXCHG instruction[22] on x86 compatible processors. No blocking is performed, the client is responsible for repeating requests if an access fails.

  Allocations are avoided by storing the most recent unused array chunk cleared by the reader instead of deallocating it. The likelihood of the chunk still being in a shared cache is high, which is beneficial for performance. Swapping pointers atomically is done with XCHG[23].

- Notification framework: The heart of ZeroMQ is its notification framework. It enables the lock-free design of independent threads which make up the concurrency model of ZeroMQ. Threads communicate exclusively via message passing, where messages are defined as commands. Each thread has a mailbox on which it listens for incoming commands. Mailbox uses Unix poll to passively listen for requests. When a signal is received, the mailbox accepts commands over it's pipe. Commands are processed by extensions of the template method[21] *object*, which defines an interface for inter-thread communication. The incoming command enumeration is inspected and the appropriate method called.

Limitations

- Thread-safety of sockets: A ZeroMQ socket can only be accessed from a single thread. A full memory-barrier must be used when migrating a socket between threads[24].

  This inherently limits e.g. multiple threads processing requests in parallel. ZeroMQ provides a solution in form of the *request-reply-broker* pattern[25] using router and dealer sockets.

- Socket-type incompatibilities: Due to the strict pattern approach of ZeroMQ, users must be aware of socket-type incompatibilities. If an application does not fit the patterns provided, custom solutions by way of mixing and matching sockets is not possible.

For example, a stream aggregation and redistribution proxy cannot be implemented. Publish-subscribe is meant only for subscription to a single stream, push and pull sockets are incompatible with router and dealer. Thus the user would be forced to resort to a solution outside of ZeroMQ, possibly still using ZeroMQ as means of transportation.

- Limited flow control restrictions: As the *high water mark* controls only the rate of messages and message size is arbitrary, it provides no effective control over the transfer rate. Furthermore, the HWM only limits the sending rate. No control of the receiving rate is provided by ZeroMQ. This can lead to a receiver allocating space for unprocessed messages until a memory overrun occurs.

  Solutions are to either enforce a maximum transfer rate on each sender with HWM, a cap on message size and limit the number of senders or implement flow control in the client.

- Inseparability of message header and payload: The ZeroMQ API provides no means to extract the payload of a received message in-situ. This limits the client architecture as a pointer to the ZMQ message must be stored and payloads must no be deallocated by themselves.

  A solution is to provide a message wrapper in the client which handles payload access and deallocation.

2.2 Serialization

2.2.1 Overview

Transferring raw objects over networks, storing them in files or memory buffers destroys the context the object was stored in. Memory addresses and byte orders change, type information is lost or invalidated and the object layout differs. The lifetime of an object is limited by the run-time of the process it resides in. Serialization is thus the process of abstracting the content of an object to make it independent of system architecture and object environment, making objects portable and persistent.

An overview of the common serialization formats XML, JSON, Apache Thrift and Google Protocol Buffers is provided by [26]. They can be divided into two general categories:

- Binary: Binary formatting encodes types into raw bytes of data. A naive way is to directly copy types as found in memory. More sophisticated approaches try to avoid storing high-order zeros and avoid storing redundant information.

  Advantages typically include CPU-efficient serialization and deserialization and compact storage.

- Human-readable: Data is encoded as strings. Fields are encapsulated with text identifiers. While being more verbose than binary formatting, text compression is often used to reduce size.

  The aim of human-readable serialization is to allow humans to directly read and edit serialized data. Especially when developing and debugging this property is useful. Use of strings provides byte-order independence.

[27] compares performance and size of binary and XML serialization formats in .Net and Java, concluding that binary formats are faster by up to an order of magnitude and use 25% to 30% as much space.

Distributed applications extensively use serialization for message passing and remote method invocation. Java serialization[28] has therefor been thoroughly investigated and many improvements suggested. In their paper[29] Opyrchal et al describe how pickle sizes in the standard Java serialization algorithm can be reduced by 50%, saving bandwidth. Although compression yields even better results, it introduces significant additional latency.

Avoiding conversion into an intermediate representation can yield speedups of up to 300%. A serialization routine is dynamically specialized according to the receiver's platform. Deserialization is thus simplified as the receiver must not unpack and reconstruct objects. Furthermore, using a hash table of references can avoid object retransmission[30].

While [31] describes many runtime and format optimizations to Java serialization, CoLoRS[32] shows that it is possible to avoid serialization altogether when sharing objects in memory between different object-oriented languages. The authors define a common class/object model, which is translated to each language runtime by intercepting field accesses. A profound restriction is that no code sharing between languages is allowed, because checking equivalence of two functions is in general undecidable. Pointers from the shared heap to a private heap are not allowed to avoid violating type safety.

2.2.2 Kryo

Kryo[33] is a serialization framework designed as a fast, efficient and simple replacement for the default Java serializer. Several default serializers for Java base types are included. Additionally custom serializers for user-defined objects can be registered dynamically. Shallow and deep copying of objects, compression, encryption and chunk encoding are supported. Forward and backward compatibility of objects can also be handled.

Kryo supports serialization of general, previously unknown class types. Registration of object serializers makes the process both faster and more space-efficient.

Registration saves an identifier for the class type in a hash map to the serializer. The ID is either specified by the user or automatically generated by Kryo. Generated IDs depend on the order in which serializers are registered, which must be taken into account by both reader and writer clients. If a class is not registered the fully qualified class name is used for identification and class member information is transmitted for reconstruction.

Custom serializers can be defined by extending the *CustomSerializer* interface. By default, if a serializer for a type does not exist it is constructed automatically. Kryo uses built-in generic serializers, e.g. array, map, collection, or uses Java reflection to retrieve a description and constructor for the class type. For most cases built-in and generated serializers suffice. With custom serializers it is possible to exert complete control, e.g. over which fields are included and how they are serialized.

Limitations

- Language interoperability: Java is the only supported language by Kryo. This extends to other languages which are built upon JVM (Scala, Clojure, etc.). Type and model incompatibilities make it difficult to support Kryo on other environments in its current form.
- Persistency: The default Kryo serializers have no facilities for extension of class types or serialization format. This limitation is alleviated by the included *TaggedFieldSerializer* in version 2, which makes field changes in classes possible. However, serialization still includes no version information for format forward and backward compatibility.

2.3 Applications for NaaS

2.3.1 MapReduce

MapReduce[34] is a concept for distributed computation of large data sets developed at Google. It is inspired by the *map* and *reduce* operations found in functional languages. The primitives are implemented by the tenant:

- Map: Takes as input a key / value pair. Outputs a set of intermediate key / value pairs.
- Reduce: Merges intermediate pairs with equal keys. The output is stored e.g. on a distributed file system or in a central database.

Input is partitioned to multiple map instances. The map jobs typically save their intermediate output on local storage. From there it is read and sorted in a distributed fashion before being passed on to the reduce instances. An optimization is to interpose a *combiner* function locally on the host executing the map job. The function performs a reduction on the local set of intermediate outputs to reduce its size. In their paper Dean and Ghemawat have noted that network bandwidth is a scarce resource and have counteracted with locality optimizations.

Restricting the computation model simplifies the implementation of parallel, distributed applications greatly. The MapReduce run-time system takes over responsibility for data partitioning and load balancing, job scheduling, failure handling and communication between jobs.

The approach has gained popularity in the industrial and academic worlds. Next to the proprietary Google implementation there exists the open-source Apache Hadoop project. Research targeting MapReduce has formalized the underlying concepts, exploited data locality on multi-core nodes, harnessed the computational power of multiple clusters to distribute work among them, applied a high-level SQL-style declarative approach to MapReduce and implemented MapReduce on GPGPUs[35, 36, 37, 38, 39].

2.3.2 Twitter Storm

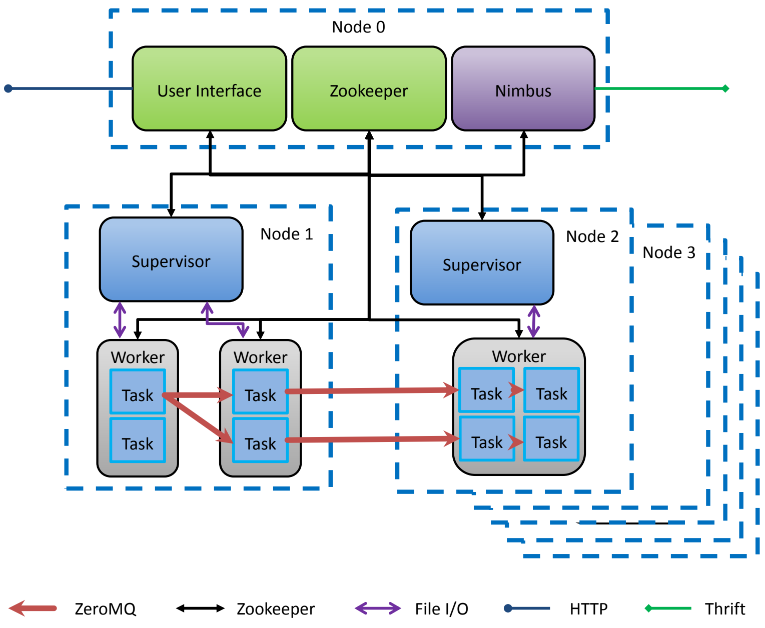


Figure 2.2: Storm Overview

Twitter Storm[16] is a distributed real-time computation system written in Clojure and Java. The project aims to fill the void left by the Hadoop and MapReduce systems. These provide general frameworks for batch processing, while Storm provides a general framework for real-time processing.

For example, Twitter processes tweets in real-time with Storm for their publisher analytics product, Groupon builds data integration systems on Storm to analyze, filter and normalize user input, and many others apply regular-expression filters on logs in real-time with Storm.

Concepts

Topology A topology is a set of *tasks*, a task being either a *bolt* or a *spout*, connected by *streams*. These can be viewed as a elements of a directed graph, where tasks represent vertices and streams represent directed edges. Because a topology is assumed to process a real-time stream, a termination criterion as such does not exist. The only way to terminate a running topology is to send the “kill” command. An exception to this rule exists for development purposes, where a timeout with subsequent termination may be defined.

For example, a spout emits a stream of generated *tuples* to a bolt, which does the first half of the processing and sends the intermediate result to a second bolt. The second bolt completes the processing and sends the final result to a third bolt, which aggregates all final results sent to it in a distributed relational database.

A parameter can be set to control the parallelism of bolts, in other words how often a bolt is replicated in the topology. For each bolt a *stream grouping* is set, which defines how the results are distributed.

Note that a topology only describes an abstract relationship between tasks. The physical layout of an active topology is controlled by the Storm scheduler.

Figure 2.2 depicts the described example topology, with all bolts having parallelism set to two.

Streams and Tuples

- A stream is defined as an unbounded sequence of tuples.
- A tuple is defined as a named list of values. The values of fields may have different types.

As mentioned previously, a stream can be split, its tuples directed to multiple bolts for parallel processing.

Every stream is assigned an identifier for selecting it from a set of streams. Every tuple is assigned an identifier, used to acknowledge a tuple for reliability. We will go into further detail shortly.

Task Tasks are concrete computational elements in a cluster. They emit and absorb streams. Spouts and bolts are summarized as tasks:

- A spouts is defined as a source of one or more streams, emitting tuples. A spout can source data from anywhere, e.g. files, databases, a non-Storm stream, etc.
- A bolt is defined as a processing element absorbing one or more streams and emitting none, one or more streams. A bolt can do any type of computation on the stream, e.g. aggregation, filtering, transformation, etc.

All tasks are assigned a component ID, which is used to identify its position in the topology, and a task ID, which is used for locating a task instance in the cluster and routing its streams to it.

Stream Grouping A stream grouping defines the method of distributing a stream's tuples among the parallel instances of a bolt. A bolt absorbing multiple streams can define a different stream grouping for each stream.
The stream groupings provided by Storm are:

- Shuffle grouping: A uniform distribution of tuples to bolts.
- Fields grouping: A field of a stream is defined as the key. Tuples with equal keys are mapped to the same bolt instance.
- All grouping: A broadcast. The stream is replicated on all bolt instances.
- None grouping: Distribution of tuples is undefined. The current Storm implementation does shuffle grouping.
- Direct grouping: The emitting task specifies the absorbing bolt for each emitted tuple.
- Local or shuffle grouping: Bolts located on the same worker as the emitting task are preferred. Equivalent to shuffle grouping for remote bolts.

By default the shuffle grouping is used. Custom stream groupings can be defined by implementing an interface.

Architecture

The components of a Storm cluster are shown in figure 2.2. We describe them one by one:

Nimbus A Storm cluster is controlled by a master process, called "Nimbus". Nimbus implements a set of Apache Thrift[40] RPC functions which read and adjust the cluster state. A command-line client exists to call these function. Alternatively, the RPC functions can be called directly from any language supported by Thrift.

The most important functionality of Nimbus is:

- Defining a topology: A topology can be defined by passing Nimbus a Java archive ("jar") containing a class which builds a topology using Storm's TopologyBuilder class. The class is a wrapper around a subset of the remote functions Nimbus implements. Thrift RPC can also be used for setting up a topology from outside of Storm.
- Starting a topology: The Nimbus scheduler keeps track of how many assignments each worker has. When a topology is started tasks are scheduled. If there are empty slots available on supervisors, tasks are assigned to those slots. Otherwise tasks are assigned to running workers, the most lightly loaded workers are given preferrence. The files belonging to the task are uploaded to the supervisors by Nimbus before starting the topology.
- Killing a topology: Nimbus first stops all spouts, waits a defined timeout for the bolts to finish processing in-flight tuples and then stops the bolts.

- Registering supervisors: When a new supervisor is started it registers itself with Nimbus. Most importantly, Nimbus stores the number of available workers on the supervisor and its IP address or domain name.
- Monitoring supervisors and tasks: Both supervisors and tasks send heartbeats to Nimbus in regular intervals. If a supervisor goes down the tasks of its workers are reassigned to another supervisor. If a task goes down it is reassigned to another worker.
- Rebalancing tasks: When the available worker pool changes, e.g. new supervisors start or go down, the workload may become unevently distributed. An explicit call for rebalancing re-distributes tasks to workers.

ZooKeeper Apache ZooKeeper stores the cluster configuration and state. For example:

- Topology, supervisor, and task information written by Nimbus.
- Supervisors and workers read their configuration from ZooKeeper.
- Task statistics on processed, in-flight and failed tuples.
- Heartbeats of supervisors and tasks.

User Interface The user interface accesses statistics and state information stored on ZooKeeper and displays them via HTTP. Controlling the cluster from the UI is not possible.

Supervisor Storm workers are managed by a supervisor local to each node. A supervisor has one or more slots, each slot specifying the port a worker listens on.

The supervisor functions as follows:

- A new worker is launched when a task is assigned to a slot which is not yet filled. The worker is assigned a generated ID, a slot (port) and its supervisor by the supervisor.
- On launch of a topology, all files necessary for launching a task are downloaded from Nimbus and copied to all directories of workers assigned to the topology.
- Running workers are monitored. If a worker heartbeat times out too often, the worker is cleaned up, i.e. the process killed if it still exists and files removed.
- ZooKeeper is monitored for changes. If there are open slots workers are started on-demand, worker configuration changes are written to file.

All communication with workers is performed via file I/O. Worker heartbeats are written in a heartbeat directory as files with increasing sequence numbers as names. Worker configuration is passed in system-specified files.

Worker Tasks are assigned to and run by worker processes. Workers have an ID, although workers are often identified by their node and port.

Every task is started in its own thread and the worker does the following:

- In a thread the worker starts a "virtual port". The virtual port binds the worker's assigned port with a ZeroMQ *pull* socket and multiplexes tuples from all incoming Storm streams to the respective tasks based on the task ID found in the tuple. ZeroMQ *pair* sockets are used for passing tuples to tasks.

  ZeroMQ is described in section 2.1.2.
- Tasks write outgoing tuples to a Java LinkedBlockingQueue[41], which is thread-safe. The worker sets up ZeroMQ *push* connections to the superset of destinations of all tasks. A thread blocks on the queue and multiplexes tuples to their destination nodes.
- The worker monitors its configuration file for changes. Tasks and outgoing ZeroMQ connections are added and removed on demand.
- A thread writes heartbeat files, as described previously.

Task A Storm task implements either a bolt or a spout interface. The worker is designed as an implicit loop, hence functions of the interface are called in a loop by the worker. The task registers the types of fields in outgoing streams with Storm. Serialization and deserialization of tuples is done with Kryo (section 2.2.2) before passing a tuple to the task and after the task has emitted a tuple.

The core functions bolts and spouts implement are:

- Bolts implement *execute*, which processes an incoming tuple passed as argument.
- Spouts implement *next tuple*, which generates a tuple.

Both call a function *emit* to send processed or generated tuples. *emit* determines the destination task via the stream grouping function defined by the topology. Tuples are then pushed onto the worker's outgoing queue.

Reliability Storm provides a reliability framework for guaranteed tuple processing:

Every tuple is given an ID. Every tuple emitted by a spout is seen as the root of a tree. When a tuple is processed by a bolt, the new result tuples are added to the tree as children. Adding children to the tree must be done explicitly by *anchoring* the tuple being emitted to another tuple. At each step traversing down the tree an acknowledgment or negative acknowledgment (fail) message is sent to a tracking task (*acker*).

If all children up until the leafs are acknowledged successfully, the whole tree is considered fully processed. Otherwise, if either a fail is sent or a tuple times out, the whole tree is considered failed and the root tuple ID is passed to a failure handler in the spout.

Limitations

- In the current Storm implementation the task scheduler does not consider locality, which could potentially reduce network traffic.
- The scheduler does not consider the processor load, memory usage, available bandwidth and other system information when allocating tasks. The sole consideration is the number of tasks running on a worker. This could lead to uneven load distribution among nodes.
- Setting up a topology from a language other than Java is cumbersome. In Java a native wrapper API with convenience features is provided, all other languages must interface the Thrift API.

Chapter 3

Design

In this chapter we present the NaaStorm architecture and both its physical and logical placement in the cluster. We reason about our design decisions and alternative choices. Some design choices are consequences of design choices made in the Storm project, others are more of practical nature. We try to give an understanding of the "why" and the "how" we constructed NaaStorm the way we did.

3.1 Data Center Placement

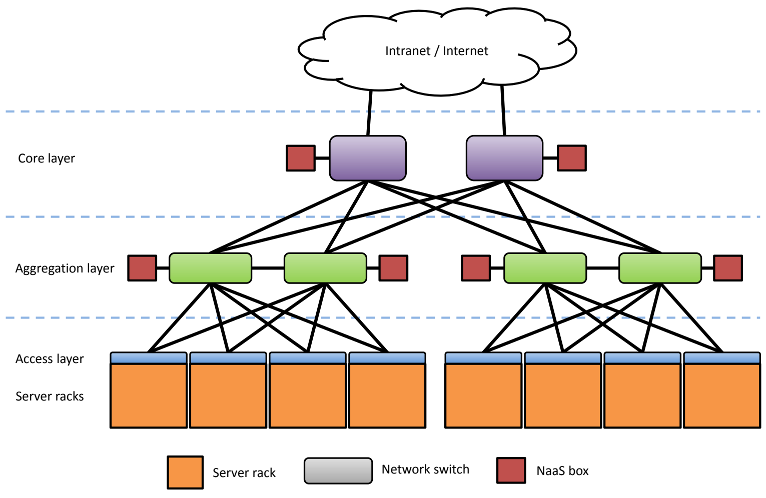


Figure 3.1: Placement of NaaStorm within Data Center Network

In our design we assumed a common data center layout as described in section **??**. Figure 3.1 shows placement of nodes running NaaStorm workers. Selected switches, preferably at higher layers, have NaaStorm nodes directly attached via high-speed interconnection (i.e. $\geq$ 10 Gib Ethernet at aggregation layer or $\geq$ 50 Gib Ethernet at core layer). This placement gives maximum throughput and equal latency to all Storm worker nodes located on leaf nodes.

Depending on the task at hand, NaaStorm nodes in larger networks may become overloaded. A measure to counteract this is to place NaaStorm workers at lower layer switches, offloading those at higher layers. Lower latency is an additional benefit of locating nodes closer to server racks.

3.2 Storm Cluster Placement

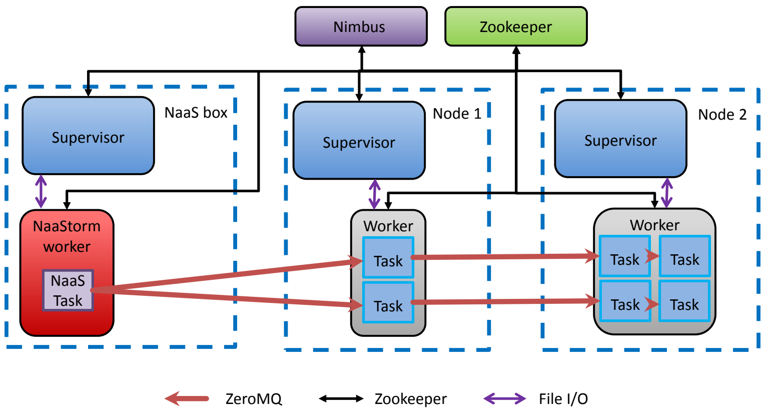


Figure 3.2: NaaStorm Overview

Integration of NaaStorm into the Storm cluster, described in section 2.3.2, can be done in several ways, keeping in mind the goal of accelerating a task and executing it on an predefined node in the cluster.

Component replacement Two levels of the Storm topology were considered for replacement:

- Supervisor: Replacing from the supervisor downwards gives full control of the node. Most importantly, the topology of supervisor, worker and task could be redefined e.g. to merge supervisor and worker. Merging of supervisor and worker would simplify monitoring, as worker heartbeats would be eliminated, and make task assignment cheaper, as we substitute inter-process for inter-thread communication. When keeping the original topology, assignment of tasks to workers and monitoring the workers could be adjusted.

  Re-implementing the supervisor requires being able to communicate with Nimbus and understanding the Storm topology information, in addition to the requirements of re-implementing a worker.

- Worker: Replacing the worker gives us full control over the implementation language, semantics of a task, the task API and data flow in the worker. Partial control over communication routing is also achieved, although re-routing breaks the topology setup by Nimbus and is best avoided.

Depicted in figure 3.2, the approach chosen is to replace the worker, since our main goal at this structural level is the acceleration made possible by control over implementation language, data flow and task API. We trade off full control of the node for a simplified design. Replacing the supervisor would yield only questionable gains in exchange for considerable additional effort.

Task to node allocation Task to supervisor mapping is defined by Nimbus in the topology state stored in Zookeeper. Again there are multiple ways to alter cluster topology, three of which were considered:

- Modification of Nimbus: Adding functionality for client-defined topology mapping to Nimbus is architecturally clean and has no ill side-effects in topology management. It requires in-depth understanding of Nimbus, as an implementation touches many areas in code. Changes to code may negatively affect stability and performance of Storm and make version updates more elaborate if they are not included upstream.

- Accessing Zookeeper: Topology state is held in Zookeeper as a struct. It can be accessed via Zookeeper's RPC interface. For changes made there to take effect, they must occur after Nimbus has written the topology configuration and before the first supervisor issues a read request on start-up. Furthermore, each time physical cluster topology changes (supervisors, tasks added or removed) Nimbus adjusts running topologies. Therefor Zookeeper must be monitored for changes issued by Nimbus and corrected when necessary.

- Packet interception: All messages relevant for task operation, encapsulated in IP packets, are necessitated to pass through precisely the switch the NaaStorm node is attached to. Deep packet inspection could be used to filter and intercept Storm messages. The clear benefit of this method is that it liberates us from modifying Storm source code.

  The method also has several unfavorable drawbacks:

  - Switch support for deep packet inspection (DPI), defined as forwarding packets based on content in addition to structured information found in the header[42], must be present. In data centers this is increasingly the case for higher layer switches[43, 44], however in testbeds it is not yet common. Lack of hardware DPI can be circumvented through software implementations. In either case DPI impacts throughput and latency of all packets processed by the switch.
  - Packet header processing for DPI cannot be done by their respective protocol libraries. Basic parsing would need to be implemented to filter Storm messages.
  - Substitution of Storm tasks is limited as the topology structure is not modified. For example, in a simple topology with two tasks, where one node counts strings sent by the other, replacing the sending spout by a in-network task would still necessitate running an idle Storm spout to construct the topology. In the described topology replacement of the bolt is also not possible, however if two bolts are chained together the first one could be eliminated entirely. Spout packets could be intercepted and results forwarded to the remaining bolt. From Storm's perspective, a task would be processed implicitly.

The first approach was chosen. While the initial implementation may be difficult, it gives us the most flexibility in setting up topologies and has the least potential for later problems.

3.3 Storm Integration

3.3.1 Worker Start-up and Heartbeats

When the supervisor launches a worker, we want to decide if it starts a Storm or a NaaStorm worker based on the assigned task. Worker heartbeats must also be solved to prevent the supervisor from timing out on heartbeats and starting new workers. Our choices are:

- Modification of the supervisor: Since the supervisor has all necessary information on hand when executing the worker, modifying it to make a decision which worker type to launch is a small change. Disabling heartbeats could be achieved by commenting out the code responsible.

- Interception or overloading of syscalls: The Storm worker is an independent process which is started with a system call. We could intercept it with a loadable kernel module[45] or overload the library it is calling[46] to start an initialization program. The initialization program would retrieve the necessary information from Zookeeper and then decide on how to proceed. It would also take over the worker heartbeats.

  Apart from being fragile, this approach is unnecessarily complicated, as we will see in the next option.

- A compromise: The path to the worker launched by the supervisor is modified in code to start an initialization program which functions as described above. Changing a path is a very small change, the main effort would lie with the initializer.

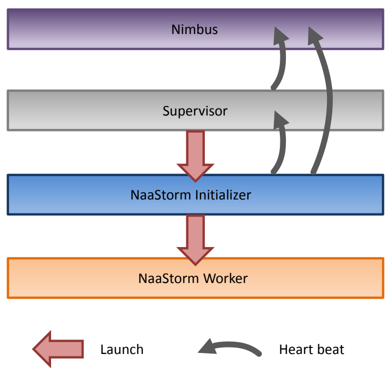


Figure 3.3: NaaStorm Process Stack

The approach chosen is the described compromise. Figure 3.3 shows the process stack of our approach starting a NaaStorm worker. The supervisor launches an initializer, which takes over the decision which worker to start. Since tasks also heartbeat to Nimbus, the initializer takes over both worker and task heartbeats if the NaaStorm worker is chosen. This relieves us from re-implementing the heartbeat functions if the NaaStorm worker is written in a different language than Storm.

3.3.2 Communication with Storm Cluster

Communication between nodes is the essence of a distributed system. Integrating into Storm means adopting Storm node's behavior in NaaStorm. NaaStorm's communications are split into following parts:

- Configuration and topology retrieval: The NaaStorm worker's configuration and topology, e.g. port number, task identifiers, destination nodes and ports, are retrieved from ZooKeeper by the initializer and passed to the worker as command-line arguments. The initializer already retrieves this data partially for its decision which worker to start, retrieving the remainder thus requires little effort.

  The data is non-persistent, using command-line arguments is simple and, in contrast to a configuration file, requires no common format if the implementation languages of Storm and NaaStorm differ. We must take care not to exceed the maximum argument length defined by the operating system. Modern versions of Linux define their argument size as 1/4 of the stack size[47]. Maximum argument length is therefor unlikely to become a problem, even with a large topology.

- Worker and task heartbeats: Described in section 2.3.2, Storm monitors tasks and workers for failures and lock-ups by listening for heartbeats. In the previous section (3.3.1) we looked at how the initializer handles heartbeats for both workers and tasks. Our approach does not monitor the status of NaaStorm workers and tasks. Heartbeats are performed solely to minimize modifications to Storm, because Storm cannot be configured to disable heartbeats. Reliability is not a concern for this project. Disregarding implementation of a robust reliability framework simplifies design and implementation considerably.

- Inter-task messaging: Streams of tuples are sent between tasks via ZeroMQ. Section 3.4.2 explains in detail how NaaStorm tasks communicate.

- Tuple acknowledgments: Storm tasks expect all sent tuples to be acknowledged within a defined time frame, either individually or in groups. Since reliability does not concern us, we do not handle tuple acknowledgments and disable them in Storm.

3.4 NaaStorm Worker Architecture

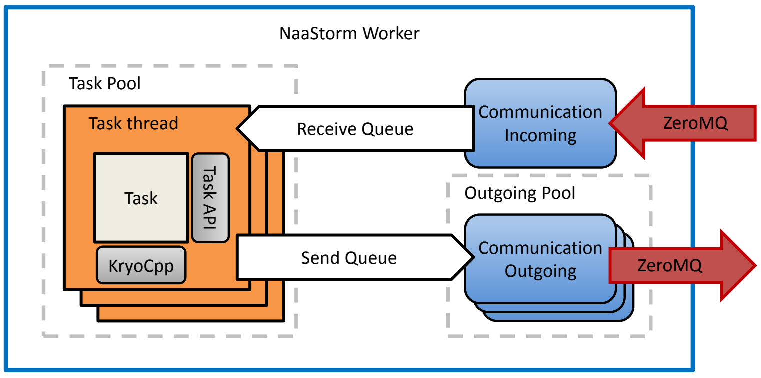


Figure 3.4: NaaStorm Worker

A NaaStorm worker consists of a thread pool embedding a task framework with client tasks, an incoming communications thread, a thread pool of outgoing communications and multiple message queues. A high-level overview is given in figure 3.4. To understand the assignments of the components, their interaction and the design choices and trade-offs made we inspect them in turn.

3.4.1 Task Framework

Clients run an explicit work loop which waits on incoming messages. The messages are processed and placed on the respective outgoing queue linked to the node of the target task. It is the responsibility of the task to know the cluster topology. Necessary topology information, such as component ID to target mapping, is provided by the task API. Message meta-data, e.g. the source task ID, is provided for each incoming tuple.

Like the Storm task API, the NaaStorm API formats messages as tuples of typed fields and handles the serialization and deserialization thereof. The task must specify the tuple type on receiving and sending a tuple. Note that dynamically changing the tuple type, e.g. sending an int and then a string, is not supported as this would require sending explicit typing information along with each tuple, incurring overhead.

Formatting messages as tuples simplifies the API and thus the client application. Hiding the implementation gives us more design choices and flexibility in the implementation and optimizations made in our framework benefit all client applications.

An explicit loop gives the client more flexibility in ordering the processing of messages. For instance, a client may want to form a window of x messages to calculate a moving average or reorder messages before processing them. It is difficult to achieve an equivalent amount of freedom with an implicit loop, in which the task framework loops over calls to abstract functions the client implements.

Explicitly selecting an outgoing message's destination relieves us from implementing stream groups as found in Storm. Most stream groupings, such as shuffle, field, all and global grouping, are trivial to implement for a client. Not including this functionality makes the task API more compact in exchange for defining the equivalent of a custom grouping in each task. The stream grouping for the destination bolt defined in the topology can be disregarded as routing tuples is the responsibility of the source task.

All tasks run in their own thread, the threads being managed in a thread pool. By giving tasks their own threads we are using the operating system's scheduler and thread control facilities. The tasks are expected to run for a long time, i.e. hours or days, marginalizing the thread start and join times.

Light-weight user-space threads present an alternative to running each task in a kernel thread. Their advantages include cheap start times and context switches, which come at the cost of additional thread scheduling complexity. Because the expected number of tasks does not exceed the number of physical processor threads and task threads have long run-times, the additional complexity in implementation exceeds their usefulness.

3.4.2 Inter-task Communication

- Incoming connections: Each worker has one TCP port assigned to it by its supervisor. The worker listens on its port for incoming ZeroMQ connections. Each arriving ZeroMQ message is read to parse the Storm message header. Most significantly, the header contains the destination task ID. The corresponding task queue is looked up in the task table and the message is queued. Multiplexing input to tasks is therefore performed by the incoming communication thread.

- Outgoing connections: A topology defines the set of destination nodes for each of its tasks. The superset of all destination nodes of tasks running on the worker is the set of nodes the worker must open ZeroMQ connections to.

  For each destination node we set up an outgoing message queue and a thread waiting to process messages on the queue. This is feasible because the expected number of destinations is small, i.e. ≤ 10. These threads are aggregated to an outgoing communication thread pool. Each thread sets up a ZeroMQ connection to its assigned destination node. Messages on the queue are encapsulated in a Storm message with header and sent on the ZeroMQ socket. Multiplexing output is therefore performed by the task.

  Multiplexing in the sending communicator is an alternative design choice. The benefit is that only one outgoing communication thread is needed, eliminating the thread pool and making memory consumption constant in regard to the number of destinations. The drawbacks are contention on the single outgoing message queue, as all tasks must access one queue, and Storm message encapsulation occurring in the task, making a clean architectural separation of task and communication more difficult.

Threading the incoming message listener and outgoing message senders enables them to block on a socket or a queue, respectively. Incoming messages must be multiplexed to tasks, which necessitates a thread listening on the worker's port.

With outgoing messages the situation is more interesting. As described in the limitations of ZeroMQ (2.1.2), sockets cannot easily be shared between threads. However, each task could open its own connections to destination nodes. While this would reduce local overhead when sending by eliminating queuing of messages, we argue that setting up and maintaining duplicate, persistent ZeroMQ connections over the network is more expensive than queuing messages locally.

All queues used in the NaaStorm worker must be thread-safe for both multiple readers and multiple writers. These constraints could be relaxed, as messages from different tasks must not be strictly ordered. Also, the only scenario occurring is multiple writers and a single reader.

The scenario of multiple readers and multiple writers does occur when we enable the client to improve throughput by concurrently processing tuples in the task. For this reason and for practicality we keep with the standard model of a concurrent queue. Note that if the serializer is not thread-safe, each thread must have a private instance.

Chapter 4

Implementation

In this chapter we present the implementation-specific details of the most interesting components in our project. We give code examples of logic and interfaces. We attempt to explain how the architecture described in chapter 3 was given a concrete face.

4.1 Rationale

The goal of the project is to reach multi-Gib/s throughput, performance is therefor an important consideration. Storm is written in Clojure and Java, we have chosen C++ for its comparatively low-level, close to metal performance characteristics, but also because the language empowers us to use the proven methods of object-oriented design.

Portability is not a design consideration as this project is purely conceived a research device. Linux on Intel x86 / x86-64 was chosen as our platform for its widespread support, although FreeBSD was considered as an alternative.

To give a high-level overview, we present the architectural diagram again:

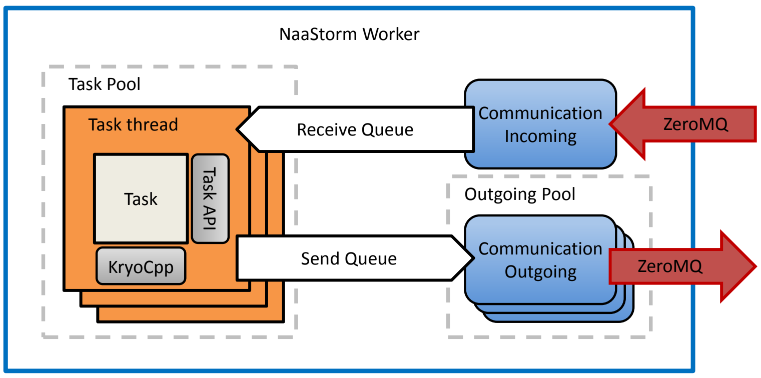


Figure 4.1: NaaStorm Worker

Note on implementation changes: Storm integration has been dropped from the scope of our project for the reasons described in section **??**. Instead, NaaStorm is implemented as a stand-alone framework. For the purpose of showing how the Storm integration in our design could be achieved we have included the described initializer.

4.2 Initializer

The initializer solves three problems:

- Decide whether to start a Storm or a NaaStorm worker depending on the assigned task.
- Retrieve the cluster information to launch a worker with a task provided by functions implemented in Clojure.

- Nimbus replaces workers if it stops receiving heart-beats form a worker. Perform heart-beating with functions implemented in Clojure.

The initializer is written in Java to access Java data structures and Clojure functions. When launched it does the following:

Command-line arguments passed in from the supervisor are parsed. These include the worker ID and port, supervisor ID, Storm ID and library paths.

The Storm configuration file is read. It contains the address and port of the ZooKeeper server. A connection to ZooKeeper is established and the cluster state is retrieved. Using the worker ID and port the map of tasks is fetched and IDs and names of tasks assigned to the worker are looked up. With the task ID map we look up the nodes, ports, task IDs and stream IDs of the worker's targets.

The task names assigned to the worker are inspected. If any is listed as being a NaaStorm task, the initializer starts a heart beat thread for the worker and each of its tasks. Then a NaaStorm worker is launched with the information retrieved previously passed as arguments. Otherwise, no additional threads are started and a Storm worker is launched, passing through the arguments received from the supervisor.

Scenarios where both Storm and NaaStorm tasks are assigned to the same node are not supported. We assume the client segregates these to different nodes.

Clojure functions are called from Java with the *clojure.lang* package:

- The Clojure packages containing the function or struct are imported
- A Clojure function is invoked by providing the package and name as arguments to a Java method "invoke"
- If the function is a query, thus has a return value, the return value's type is void. It must be cast to the expected type. Since we initially do not know which type to expect, we first deduce this by observing the dynamic type of the return value. Then a cast to the evaluated type can be applied.

4.3 Command-line Parser

Before starting a worker must know details about its assignment. The initializer provides these details and communicates them to the worker via command-line arguments. The parser is called immediately after the worker is executed.

Important arguments are:

- Task names with IDs to be run on the worker.
- Targets consisting of the task ID, host name or IP address and, optionally, port.
- Target task ID to component ID mappings.

To perform the parsing we use the Boost[48] Program Options library.

4.4 Communication Framework

Divided into *client* and *server*, the communication framework interfaces the ZeroMQ library. Both implement a *run* member function, which is executed in a Boost thread. The client is passed a reference to an outgoing queue, the server is passed a reference to the incoming queue.

Boost threads are wrappers around the native system's thread libraries, e.g. Posix threads on Linux. While this makes it more simple to port the NaaStorm worker to other systems, the main reason for using Boost threads over Posix threads is their support for C++ member functions.

The client connects to an address and port passed to it on construction. It sets up a ZeroMQ context and push socket. The high water mark is set and then a connection is established.

From then on the communication client blocks on its outgoing queue. If a ZeroMQ message is in the queue, the message is retrieved and sent on the socket.

The server listens on an interface and port passed to it on construction. Like the client it sets up a

Listing 4.1: Listen loop in the communication server

```
while(true) {
  msg = new Message();

  for (more = true, i = 0; more; i++) {
    zmqMsg = new zmq::message_t();

    rc = socket->recv(zmqMsg);
    socket->getsockopt(ZMQ_RCVMORE, &more, &moreSize);
    if (rc && (i > 0 || zmqSocketType != ZMQ_ROUTER))
      bytesReceived += zmqMsg->size();

    if (zmqSocketType == ZMQ_ROUTER && i == 0) {
      // save client ID
      msg->setOrigin((char *)zmqMsg->data(), zmqMsg->size());
      delete zmqMsg;
    }
    else {
      // save message (may have multiple parts)
      msg->push_back(zmqMsg);
    }
  }
  // multiplex packets
  target = BufferHeader::target(msg->begin());
  queue = taskidToQueue[target];
  if (queue != 0) // if queue for target exists, continue
    rcvbuf[queue].push(msg);
  else
    delete msg;
}
```

ZeroMQ context and pull socket.

The communication server then blocks on the listening port. When a ZeroMQ message arrives its header is parsed to retrieve the target task ID. The ID is the key to the target task's queue saved in a hash map. The message is pushed onto the corresponding incoming queue. Listing 4.1 shows the server listening on the socket.

Client and server support arbitrary ZeroMQ socket types, although NaaStorm only utilizes push and pull, as well as multi-part ZeroMQ messages and ZeroMQ IDs. The ZeroMQ ID is located in the first part of a multi-part message and is used to identify the sender or specify the receiver when using a router socket.

ZeroMQ messages are abstracted by the *Message* class. It hides the complexity of dealing with the ZeroMQ interface:

- Only reference is needed when dealing with messages. The distinction between ZeroMQ message pointer and pointer to the payload is eliminated.
- Templates are used to type the payload data. A type is specified on retrieving the payload and the data is structured and cast accordingly.
- Multi-part ZeroMQ messages are hidden. The data of all message parts is exported as an array of arrays.

- STL-like iterators to message parts are provided.
- The cumbersome ZeroMQ interface for destructing payload on sending ZeroMQ messages is hidden. The message class provides a generic destructor functor. If required a custom destructor can be set.

4.5 KryoCpp

Kryo (section 2.2.2) is the standard serializer Storm uses for flattening tuples before sending them as ZeroMQ messages. Unluckily for us, Kryo is a Java-only serializer and is not available for C++. To work around this limitation we devised our own Kryo-compatible serializer, KryoCpp.

Our example applications, e.g. multi-media byte streaming and wordcount, use only a subset of Java's basic data types and data structures. Hence our intent was not to construct a fully-fledged re-implementation of Kryo, but a high-performance, compatible serializer for the necessary types.

4.5.1 Serialization of Important Data Types

Before describing the implementation, we must understand how Kryo serializes types. Recall that Kryo serializes from and to a Java ByteBuffer. Important are:

- byte: bytes are written into the buffer in raw form.
- int: Two modes are defined for serializing integers: standard and optimize positive. Standard mode divides the range between positive and negative numbers equally, whereas optimize positive gives priority to positive numbers.

  An encoded integer has a maximum length of 5 bytes. 7 bits go into a byte - the highest-order bit of each byte determines if there is a next byte. If all remaining high-order bits of the integer are 0, encoding stops.

  Standard mode places the sign bit at the lowest-order bit of the first byte. If the number is negative, it is bit-wise inverted before further encoding.
- array: The first byte contains the number of dimensions of the array. The first dimension immediately follows. Each dimension is started with an integer storing the size of the current dimension. If the last dimension has been reached, the elements are deserialized by the respective method. Else the next dimension is processed.

  Some values can be predefined to optimize space and run-time:

  - Number of dimensions.
  - Sizes of dimensions. This also specifies the number of dimensions.
  - All elements have same type. Type information must then only be stored for the first element.
- string: An integer at the front specifies the number of characters in the string. The characters are then read and a string object constructed.

  Java supports UTF-16 by way of representing supplementary characters as surrogate pairs[49]. Each UTF-16 character is encoded into two or three bytes, ASCII characters are encoded with up to two bytes. The highest-order bit in all bytes specifies single- or multi-byte encoding. In case of multi-byte, the four highest-order bits of the first byte specify if the character is two or three bytes long.

Before writing an object to the buffer, Kryo pushes a byte specifying if the object is or isn't *null*. If the object is *null*, 0 is pushed and the serializer returns, otherwise 1 is pushed and the object is written.

Listing 4.2: KryoCpp public interface

```
class KryoCpp {
public:
  KryoCpp();
  virtual ~KryoCpp();

  template<class T, class InIter = InputIterator,
    class OutIter = OutputIterator>
  static size_t writeObject(OutIter& outiter, T const& object);

  template<class T, class InIter = InputIterator,
    class OutIter = OutputIterator>
  static int readObject(InIter& initer, T& object);

  template<class T, class InIter = InputIterator,
    class OutIter = OutputIterator>
  static T readObject(InIter& initer);

  template<class T>
  static size_t getBound(T& object);
}
```

4.5.2 Interface

The client creates an instance of the KryoCpp class. A STL byte queue, passed by reference on each call to the serializer, is used for storing serialized values. The public interface accessible to clients is shown in listing 4.2.

The functions *readObject* and *writeObject* read and write serialized objects to and from the buffer. The client must specify the expected type explicitly when reading via return value. This is unnecessary when writing or reading via reference as the type of the parameter passed is known. The buffer is accessed exclusively with iterators to support different underlying buffer types. Input and output iterators are assumed to conform to the STL forward iterator type.

Single-dimensional arrays are supported via std::vector and std::array. Vectors can be dynamically re-sized, thus they can be used for arrays of arbitrary length defined at run-time. Alternatively C++11 std::array can be used. However, the length must then be defined at compile-time.

Multidimensional arrays are not supported as they exceed our requirements and there are multiple ways Kryo serializes them.

getBound returns an upper bound for the serialized size of a specified object. This is useful to determine the necessary buffer size to store an object.

KryoCpp is not thread-safe. While parallel operations on two unrelated buffers are supported, operations on the same buffer are not safe as multiple subsequent accesses, required for several data types, are not performed atomically.

4.5.3 Internals

The basis of KryoCpp is the bridge pattern, described in [21]. The *KryoCpp* class is an abstraction of the implementing *Serializer* class. The is-null Boolean byte is handled in *KryoCpp* before calls to *Serializer*.

Serializer provides two template function stubs *get* and *put*. These are the pendants to *readObject* and *writeObject* and they are implemented by overloading the template functions[50]. For example, the implementation of *get* for integers is shown in listing 4.3.

The functions *getArray* and *putArray* process the array dimension before calling *getSequence* and *putSequence*, respectively. These retrieve or push a sequence of given type on the queue and are also used by the string serializer.

All API functions are defined as static to avoid constructing a KryoCpp object and passing it in on each call. Some serializers, e.g. for value of type int, can be configured with constant member variables and can therefore not be static. However, these functions are in-lined by the compiler, the variables substituted for their constants and no objects are created.

Run-time serializer registration as in Kryo is not possible in the current implementation of KryoCpp as

Listing 4.3: KryoCpp integer serializer

```
int Serializer::get(InIter& initer, int& object) {
  int offset;

  object = 0;
  for (offset = 0; offset < 32; offset += 7) {
    uint8_t byte = *initer;
    ++initer;

    if ( (byte & 0x80) != 0) {
      object |= (~0x80 & byte) << offset;
    }
    else {
      object |= byte << offset;
      break;
    }
  }

  if (!optimizePositive)
  object = (((unsigned int) object) >> 1) ^ -(object & 1);

  return 0;
}
```

we only need support for basic data types. However, user-defined class serialization could be done by implementing the function template stubs of *get* and *put* as fallback for the overloaded versions. These generic functions would then check the run-time type of the template parameter and search for the type in a hash table of serializers. If the key is found the serializer could be called, otherwise an error returned.

4.6 Task

The task is the most important component in NaaStorm. It is the framework the client works with and the NaaStorm worker's core. A task consists of the *Task*, *TaskInfo* and *TaskTools* classes, the task interface and the KryoCpp serializer described in section 4.5.

4.6.1 Interface

A nstask, be it a bolt or a spout, implements the abstract, virtual *run* function of the *TaskInterface* class shown in listing 4.4. A *TaskTools* instance is provided for use by the client. The *TaskInfo* object provides information about the task itself and the topology.

As only the *Task* class must have access to *run* and the fields, it is a friend class of *TaskInterface.*

Tuples are defined as STL tuples, introduced by C++11. Alternatively a tuple can be a STL array, as these can also be accessed via the STL tuple interface. Both shall simply be referred to as "tuple" from here on unless otherwise specified.

TaskTools handles all communications with the cluster. It it are defined the *absorb* and *emit* functions. *absorb* retrieves a tuple and returns an integer specifying if a flush has taken place. *emit* takes a task specifier and a tuple. Tuples are always retrieved and passed by reference to avoid memory copying. The source task ID of the most recently absorbed tuple can be retrieved by calling *getSource.* Tuples are

Listing 4.4: Task interface

```
typedef std::pair<int, int> TaskSpec; // <task id, queue id>

class TaskInterface {
  friend class Task;
public:
  TaskInterface() { }
  virtual ~TaskInterface() { }

protected:
  virtual void run() = 0;

  TaskTools *tasktools;
  TaskInfo *taskinfo;
};

class TaskTools {
public:
  template<class T>             int absorb(std::tuple<T>& tuple);
  template<class T, size_t N> int absorb(std::array<T, N>& tuple);

  template<class T>             void emit(const TaskSpec target,
    std::tuple<T>& tuple);
  template<class T, size_t N> void emit(const TaskSpec target,
    std::array<T, N>& tuple);

  template<class T> void flush(const TaskSpec target);
  int getSource();

  TaskTools* copy();
};

class TaskInfo {
public:
  int id;
  std::string name;
  std::multimap<int, TaskSpec > componentidToTaskspecifier;
};
```

buffered internally, this buffer can be flushed by calling *flush* with a task specifier. After absorbing the last tuple of a flushed buffer, thus the buffer is empty, *absorb* returns 1 instead of 0.

copy returns a new instance of *TaskTools*. In a multi-threaded task each worker thread must have its own *TaskTools* object for thread-safety.

These functions could have been provided in the task interface itself by way of the template method pattern[21]. However, overriding of the methods must be prevented as, by way of the information hiding principle[51, pp. 51-53], the client should not have direct access to the interfaces necessary for re-implementation or overloading.

TaskInfo contains the task's own ID and name and a mapping from component IDs to task specifiers. When defining the topology in Storm, the component IDs of the next components is set. Hence the component IDs specifying the next bolts is known before run-time. At run-time the task must find instances of these bolts. As all tasks are identified by their unique task ID and must run in a worker on a node, they are specified by their task ID and the queue ID of the outgoing queue directed at the hosting worker. These are provided by the task specifiers retrieved from the mapping.

4.6.2 Internals

The *Task* class is defined as a functor. Its objective is to finalize construction of the client's NaaStorm task (which we dub "nstask") and to provide a compact interface to launch a task. The design is modeled on the proxy and builder patterns[21].

Starting a task Tasks are stored in a task registry by mapping a nstask's name to a wrapper around its constructor. On launch the worker receives a task name on the command-line. The given name is looked up in the registry and a concrete instance of the nstask is constructed.

The *Task* functor is started as a Boost thread. On construction it takes pointers to the incoming queue and all outgoing queues to nodes hosting its task's messaging destinations, as defined in the topology. A pointer to a concrete instance of the task interface, the nstask, and task ID are also required for creation. A new *TaskTools* object is instantiated and the queues and task ID passed to it. A *TaskInfo* object with set fields is received by a member function and assigned to the nstask instance.

Lastly, the run method of the nstask is called, completing the start of a task.

Interacting with the Cluster *absorb* and *emit* in *TaskTools* each have two interfaces: one for STL tuples and one for STL arrays. Both versions are adapters to a private, template implementation.

Serialized tuples are stored in a buffer, each buffer beginning with a message header. This is done for performance reasons to which we will come back later.

When emitting a tuple it must be placed on the specified target queue. Each target queue has an associated buffer along with a buffer iterator at the current position and a tuple counter. These are loaded and the buffer is checked if enough space is available for the current tuple via the upper bound retrieved from KryoCpp. If so, the tuple is serialized and the tuple counter is incremented by one. If not, the message header is written to the buffer, the buffer is placed on the specified outgoing queue. A new buffer is allocated and the iterator is updated.

Flushing loads the specified target buffer and the header is written to the buffer. In the header the flush bit is set and the buffer is placed on the appropriate outgoing queue. A new buffer is allocated and the buffer iterator is set.

On absorbing a tuple checks are made to ensure that a buffer exists and the buffer's end has not yet been reached. If so the tuple is deserialized and the tuple counter is increased by one. If no buffer exists, the a buffer is popped off the incoming queue. The buffer iterator and end sentinel are set and the source is retrieved from the message header. If the end of the current buffer has been reached, the buffer is deleted and the header checked if a flush has been issued. If so the buffer struct is reset to initializing values the function returns with a return value indicating the flush. If not a new buffer is fetched as described previously.

Retrieving the source task ID of the most recent tuple load and returns the source field from the buffer struct.

Tuples may have arbitrary length and types, posing the question of how to serialize them. Because the tuple type is known at compile-time and the KryoCpp functions are either overloading or templates, the serializer conforming to the type is chosen by the compiler. For this to happen the tuple must be broken down into its components. Via template meta-programming the tuple fields are iterated and passed to KryoCpp.

The same technique is used for deserialization and bounds checking.

The message header is formatted as eight bit fields, the least significant indicating a flush, four bytes encoding the source task ID and four bytes encoding the target task ID. Multi-byte values are stored in network byte order.

Performance tuning To enable the compiler to inline functions all definitions are stored in C++ header files.

The first bottleneck we hit were the send and receive function calls to ZeroMQ. Because the ZeroMQ library does not define them in the header the compiler cannot inline them resulting in a function call for each transferred tuple. Combining multiple tuples in a buffer and transmitting the buffer removes calls to ZeroMQ as bottleneck. Additionally it reduces contention on the concurrent queues, cache misses are reduced through spacial locality and message header overhead is marginalized.

The variables accessed on absorbing or emitting are assembled to structs to ensure close spacial locality. These are stored as class member fields and on accessing them the implicit "this" pointer is dereferenced. Profiling identified this as the most time consuming factor of a simple emit loop. Dereferencing on each emit is avoided by caching the most recently used buffer struct in a static variable. Caching increases performance by 50%, but brings a number of problems with it:

- Static variables are private to a function, but globally only one instance of the variable exists over all class instances. In a worker running one single-threaded task *absorb* and *emit* are exclusively called from the same context, the task loop, and the cache is private to the task. However, multiple tasks running in parallel and multi-threaded tasks will cache different buffer structs, corrupting the cache. Substituting the global static variable with a thread-local static variable makes the cache private to each task thread.

- *flush* requires the current buffer state, but does not have access to *emit*'s local cache. Writing back the cache after each emit defeats the performance gain, thread-local static class variables for non-trivial types are not yet supported by GCC[52]. By adding a Boolean in the function interface which forces a write-back when set *flush* gains access the current state.

- The cache is invalidated when the task specifies a new target. To detect the switch the current target queue ID is stored alongside the buffer struct. Switching targets triggers a write-back and an update of the queue ID. Initially the queue ID is set to an invalid value to force loading the cache.

- To measure performance a recent state of the tuple counter must be accessible. In an attempt to compromise performance and accuracy a write-back of the buffer struct occurs each time a new buffer is allocated.

Absorbing tuples is tuned similarly, differentiating only in that a target switch is not required. Note that the write-back on fetching a new buffer is necessary both for the performance measurement and for keeping the source task ID updated outside of the cache, saving an explicit write-back on each call to *getSource*.

Further tuning includes substituting passing by value with passing by reference, passing branch prediction information to the compiler for specific branches and ordering struct members by their access order.

Thread-safety Running multiple tasks in a single worker and multi-threaded tasks require the task interface to be thread-safe. In our high-level design we have already taken into account the necessity of concurrent queues, enabling the queuing and dequeuing of tuples by multiple threads.

The task interface is modeled on having no points of contact between the threads, barring the queues. This both simplifies the threading model and increases performance as no contention occurs outside the queues. Each thread has thread-local buffers, associated iterators and counters. Care must be taken to not circumvent this, e.g. with static variables as caches as described above or when multi-threading a task. When threading a task the client must explicitly construct a new instance of *TaskTools* for each thread accessing buffers to maintain thread-safety by calling the *copy* function.

copy constructs a new *ThreadTools* object while retaining all information from the current task, e.g. the queue references and task ID are copied to the new object, but new buffers and iterators are allocated.

4.6.3 Assembling the Pieces: An Example

To show how the task interface is used to implement a task, we provide an example of an incrementer bolt in listing 4.5. The bolt absorbs an integer, performs an increment by one and emits the result as a shuffle stream grouping to all tasks with component ID 0.

The task inherits the *TaskInterface* and implements the abstract *run* function. The target component

Listing 4.5: Implementation of an incrementer bolt

```
class IncrementBolt: TaskInterface {
  virtual void run() {
    const int component = 0;
    tuple<char> tuple;
    int rc;

    // get targets and cache target specifiers
    auto tasks = taskinfo->componentidToTaskspecifier.equal_range(component);

    // absorb tuples, emit as shuffle stream grouping
    while(true)
      for (auto iter = tasks.first; iter != tasks.second; ++iter) {
        do
          rc = tasktools->absorb(tuple);
        while (rc == 1); // if flushed repeat absorb
        ++get<0>(tuple);
        tasktools->emit(iter->second, tuple);
      }
  }
};
```

ID is set to 0 and the tuple is defined as a single *char*.

Before entering the work loop we retrieve a range of task specifiers with the defined component ID, our targets. The range is given via a pair of iterators defining the beginning and end.

Inside the work loop a second loop iterates over the range of targets. For each target we absorb a tuple and check the return code. If the buffer has been flushed we have not absorbed a tuple and retry. Otherwise we increment the tuple by one and emit it to the current target.

Performance could be improved by precaching the target specifiers in an array, as e.g. the GCC STL returns the target range as a tree and the retrieving function is not in-lined.

To make use of target caching another loop could be added to send a predefined number of tuples to the same target before switching to the next target.

Since buffers are rarely flushed, a branch prediction hint could be added to codify this knowledge.

Chapter 5

Evaluation

In this chapter we present the performance measurements and interpret their meaning. In the Discussion section we review the problems we encountered while designing and implementing our project, the limitations our project has and how these limitations could be resolved.

5.1 Performance

The test hardware and software configurations are:

Dell r710 (d710)

- 1 x Intel Xeon E5530 @ 2.40 GHz
- 12 GiB memory
- Ubuntu 12.04, Linux kernel 2.6.38.7-1, x86_64

Dell r820 (d820)

- 4 x Intel Xeon E5-4620 @ 2.20 GHz
- 128 GiB memory
- Ubuntu 12.04, Linux kernel 3.2.0, x86_64

Software stack

- GCC version 4.6.3, CXX_FLAGS="-std=c++0x -O3 -g -Wall"
- Java version "1.7.0_03", OpenJDK (IcedTea7 2.1.1pre)
- ZeroMQ version 2.2.0
- Kryo version 1.04
- Intel TBB version 4.0_20120408

5.1.1 KryoCpp

In this section we measure the performance characteristics of the Kryo and KryoCpp serializers.

Test Setup

The test setup consists of a loop serializing and deserializing items in the same loop iteration. The purpose of this is to avoid distortion of results through cache misses. The items are pregenerated before the loop is entered and saved in an array for spacial locality.

KryoCpp is tested with a selection of underlying data-structures:

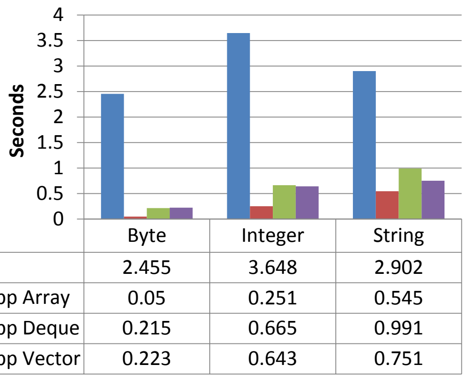
- STL Deque: a double-ended queue
- STL Vector: an array with support for dynamic re-sizing
- preallocated array

Following data types are tested:

- Java byte and C++ uint8_t
- Java and C++ int
- Java and C++ string of 6 characters. Here we store pointers to strings in the C++ array to mirror Java's references. We explicitly avoid spacial locality by introducing a layer of indirection in C++ as locality would distort results. We also avoid non-ASCII characters in Java as KryoCpp does not support UTF-16 (double byte) representation.

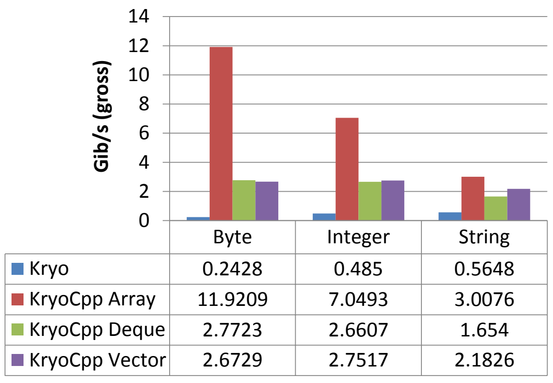
The tests are performed in a best of 5 scheme on array lengths of $4 * 10^7$ for bytes and integers, $2 * 10^7$ for strings. We assume a uniform distribution over the ranges of the respective data types.

Results and Interpretation



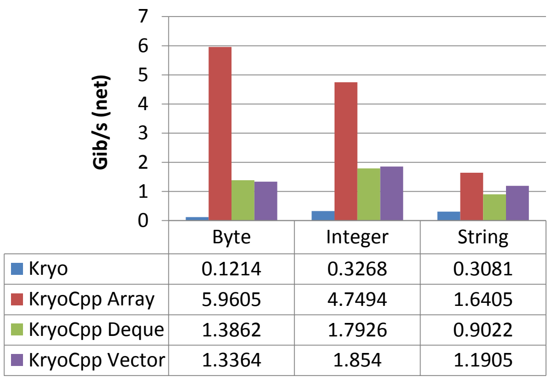
	Byte	Integer	String
🟦 Kryo	2.455	3.648	2.902
🟥 KryoCpp Array	0.05	0.251	0.545
🟩 KryoCpp Deque	0.215	0.665	0.991
🟪 KryoCpp Vector	0.223	0.643	0.751

(a) Time



	Byte	Integer	String
🟦 Kryo	0.2428	0.485	0.5648
🟥 KryoCpp Array	11.9209	7.0493	3.0076
🟩 KryoCpp Deque	2.7723	2.6607	1.654
🟪 KryoCpp Vector	2.6729	2.7517	2.1826

(b) Gross Throughput



	Byte	Integer	String
🟦 Kryo	0.1214	0.3268	0.3081
🟥 KryoCpp Array	5.9605	4.7494	1.6405
🟩 KryoCpp Deque	1.3862	1.7926	0.9022
🟪 KryoCpp Vector	1.3364	1.854	1.1905

(c) Net Throughput

Figure 5.1: Evaluation of Kryo and KryoCpp performance

Figure 5.1a shows the time spent for serialization and deserialization, figures 5.1b and 5.1c show the throughput. Gross throughput is the output data rate, net throughput is the input data rate.

KryoCpp shows the largest gains with small data types. The speedup of byte serialization with Deque is 11.418 in comparison to 2.928 for strings of 6 characters. When preallocating the buffer our gains are even greater with a speedup of 49.100 and 5.325 for byte and string, respectively.

Our reasoning for this is the structure of KryoCpp: function overloading avoids a hash map look-up of the type-specific serializers at run-time. The compiler inlines functions and in some cases uses SIMD instructions. Furthermore, when using arrays memory can be preallocated. All these factors make the serializer very efficient.

The performance gradient from bytes to integers to strings can be explained by the serializer's complexity. On x86_64 the byte serializer consists of two move instructions, whereas the string serializer performs a function call for each character. The character serializer is comparatively large and has many branches. Nevertheless we can still see substantial gains by implementing algorithms machine-natively.

Best-case bandwidth is achieved with bytes at 5.960 Gib/s net and 11.9209 Gib/s gross. Worst-case bandwidth is at 1.6405 Gib/s net and 3.0076 Gib/s gross with strings. All results support our goal of multi-Gib/s throughput on high-performance hardware.

5.1.2 ZeroMQ

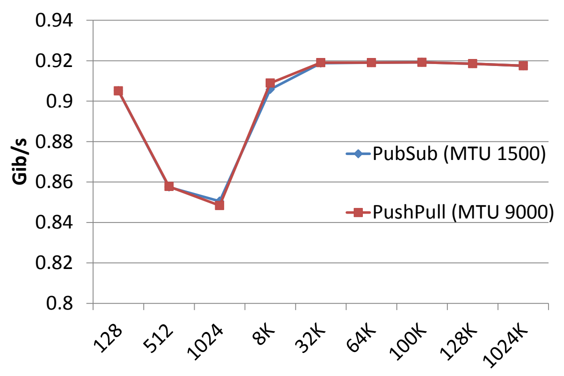
In this section we measure and compare the throughput of ZeroMQ and TCP/IP.

Test Setup

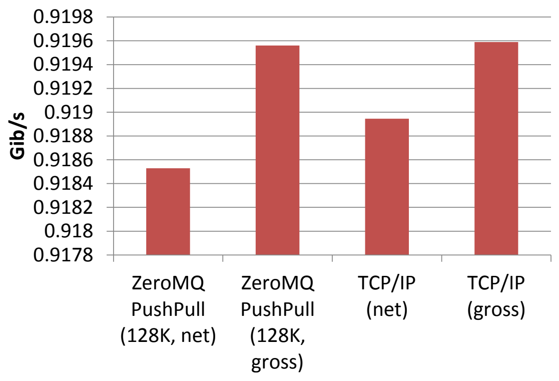
There are two test setups: two d710 nodes with 1 Gib/s network links and two d820 nodes with 10 Gib/s network links. We measure TCP/IP throughput with the *Iperf* benchmark suite and ZeroMQ throughput with the ZeroMQ benchmark suite. For closer comparison and evaluation of the overhead the gross throughput of ZeroMQ and TCP is measured with *bwm-ng* at OSI layer 2. The ZeroMQ suite is modified to use Push and Pull sockets instead of Publish and Subscribe sockets. The standard implementation is included as a reference.

The d820 nodes are bench-marked with MTU sizes of 1500 and 9000 bytes.

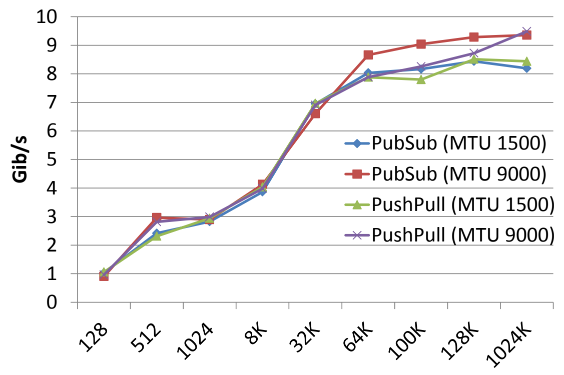
Results and Interpretation



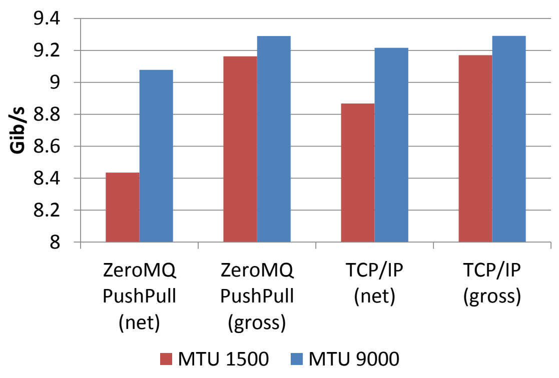
(a) Push/Pull vs Pub/Sub on d710



(b) ZeroMQ vs TCP/IP on d710



(c) Push/Pull vs Pub/Sub on d820



(d) ZeroMQ vs TCP/IP on d820

Figure 5.2: Evaluation of ZeroMQ performance

ZeroMQ shows net throughput almost equal to TCP/IP up to 1 Gib/s. With transmission rates faster than 1 Gib/s small messages sizes prove to be a bottleneck. Although ZeroMQ batches messages smaller than 255 bytes[53] before sending them over the network, the overhead of calling a function for every transmitted message and the batching process is significant. For 64 KiB messages and larger the throughput is again near that of TCP/IP. The speedup of TCP/IP throughput in comparison to ZeroMQ is $<$

1% at 1 Gib/s and 4.19% respectively 5.68% for MTU sizes 1500 and 9000 at 10 Gib/s. Note that the gross throughput of both protocols is the same.

Near 10 Gib/s we found that performance is CPU-constrained. Profiling identified the *tcp_recvmsg* kernel function, a part of the Linux IP stack, as culprit. The bottleneck is alleviated by using jumbo Ethernet frames. For an equivalent amount of transmitted data the jumbo MTU size of 9000 bytes versus the standard 1500 bytes reduces the packet processing overhead as less packets transverse the networking stack. The net TCP performance graphs in figure 5.2d give evidence.

At 512 and 1024 bytes on d710 nodes, pictured in figure 5.2a, we observe an anomaly. As the d820 measurements show no such anomaly and the software stacks are the same except for the kernel, it is most likely due to a Linux kernel regression or a hardware limitation.

In figure 5.2c we see that Publish / Subscribe sockets outperform Push / Pull sockets for message sizes 64K, 100K and 128K. Publish / Subscribe is defined to drop packets when the high water mark is reached whereas Push / Pull provide reliable transport. Although no packets are dropped in the benchmark, the throughput deviation is likely due to additional overhead in the latter scenario.

5.1.3 NaaStorm

Test Setup

In this section we measure the performance characteristics of the NaaStorm worker. We contrast the NaaS topology to the regular cluster topology and test how our implementation scales with high-performance hardware.

We have three setups:

- Standard topology: 5 NaaStorm workers running on d710 nodes are connected via a switch with 1 Gib/s links. One node is assigned the task of distribution or aggregation point.
- NaaS topology: a NaaS box is connected to 4 leaf nodes via direct 1 Gib/s links, totaling 4 Gib/s. The NaaS box is the distribution or aggregation point. The NaaS box and leaf nodes consist of d710 nodes.
- High-performance reference: 2 d820 nodes, connected by a 10 Gib/s link, run one worker each. They are included to evaluate the throughput limitations of a NaaStorm worker.

The benchmarks performed are:

- Byte streaming: We stream a set of random bytes from one or more spouts to one or more workers. The bytes are stored in an array which is iterated multiple times. The bolts receive and discard the data.

  This scenario occurs when data can be efficiently retrieved by the spout and that the bolts are not CPU bound such that the application is network bound. The first assumption is reasonable if the stream is received at another network interface or a local, special-purpose device and dumped into main memory. The second assumption is valid if there are enough bolts in the cluster providing processing power for the application at hand.

  The purpose of this benchmark is to show the maximum throughput of a NaaStorm NaaS box. We demonstrate both multi-threaded spouts and multiple spouts in a single NaaStorm worker instance. Tuples are made up of a single byte to show that both KryoCpp and NaaStorm efficiently handle small tuples.

  As proof-of-concept we add benchmarks of dual-threaded spouts to show the feasibility of threading NaaStorm tasks.

- Distributing word count: We send a stream of strings from a spout to multiple bolts counting the words. The strings are predefined and are stored in an array, which is iterated multiple times. Each word is a key in a hash map containing word counters. On receiving a word the bolt looks up the respective counter and increments it by one.

  String comparisons are CPU-intensive operations, thus in this benchmark we show the behavior of NaaStorm when the task at hand is limited by the CPU power or memory throughput of bolts.

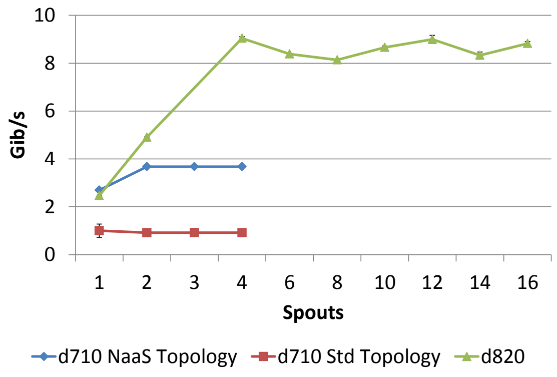
- Aggregating word count: We aggregate multiple streams of strings sent by multiple spouts at a single node. The setup is otherwise equivalent to the distributing word count.

Here we show the throughput of NaaStorm if the aggregation point, e.g. the NaaS box, is bound by CPU power or memory throughput.

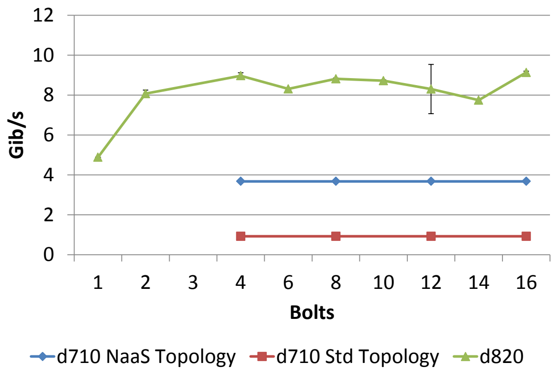
- Local data transfer: We demonstrate the raw performance of a single NaaStorm worker by sending random, single byte tuples over the loop-back interface and receiving them in bolts running in the same worker instance as the spouts.

In each benchmark we increment the number of spouts and bolts until a plateau or the maximum network throughput is reached. Benchmarks are run for 10 seconds, excluding a warm-up and shut-down period to start and stop the cluster. Measurements are taken with the *bwm-ng* monitoring tool at the distribution or aggregation point. As such the measurements represent the gross network throughput. Results are presented as the statistical mean with standard deviation.

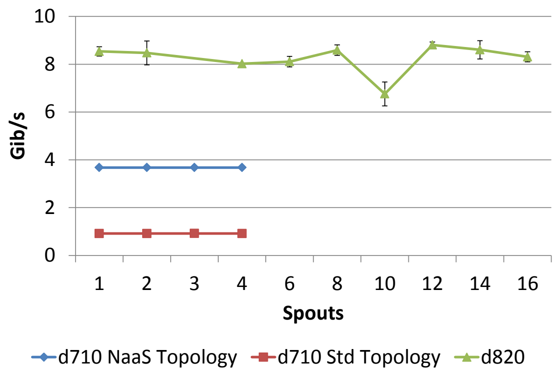
Results and Interpretation



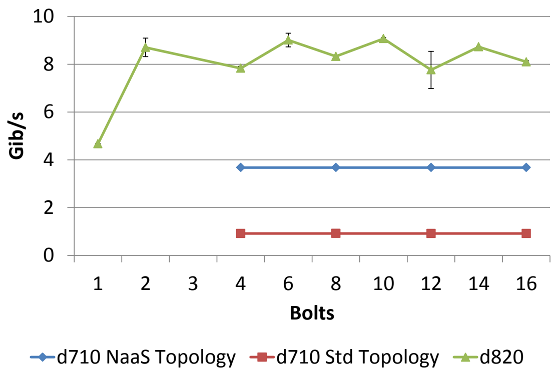
(a) Byte stream - spouts



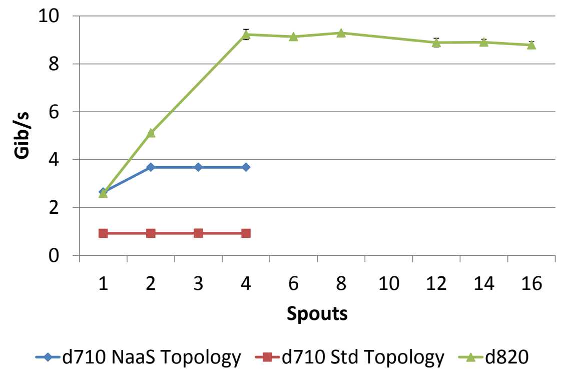
(b) Byte stream - bolts



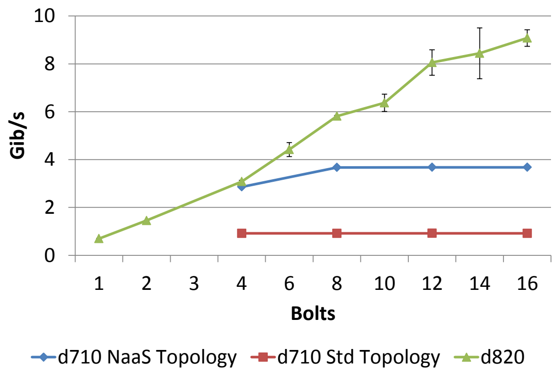
(c) Threaded Byte stream - spouts



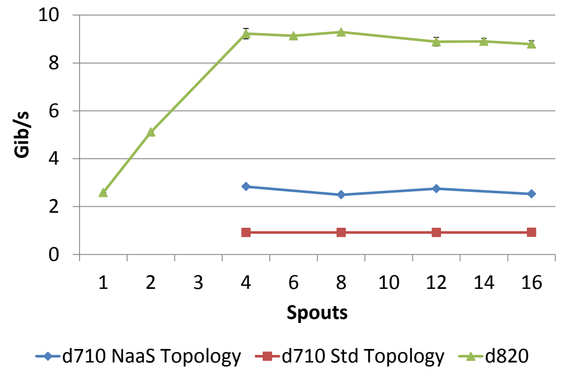
(d) Threaded Byte stream - bolts



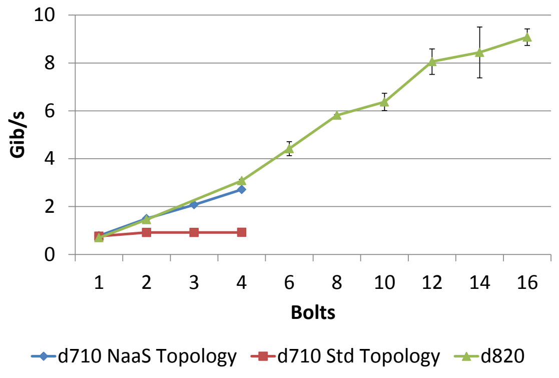
(e) Distributing word count - spouts



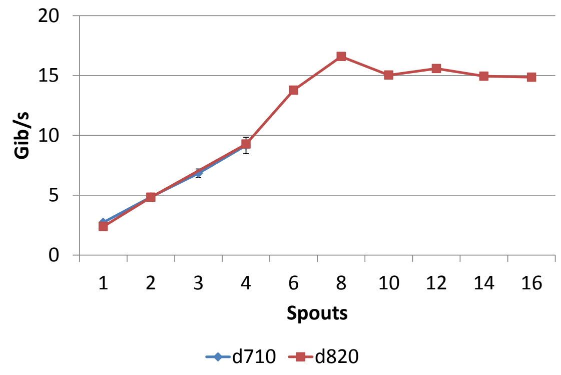
(f) Distributing word count - bolts



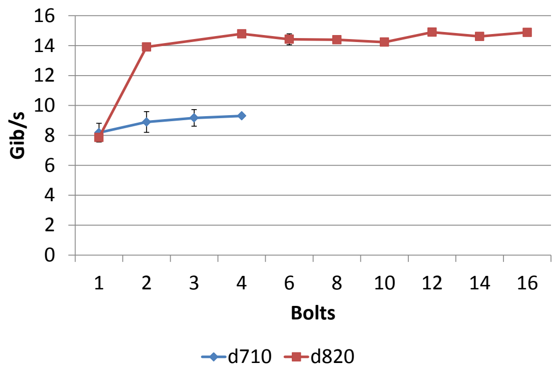
(g) Aggregating word count - spouts



(h) Aggregating word count - bolts



(i) Local data transfer - spouts



(j) Local data transfer - bolts

Figure 5.3: Evaluation of NaaStorm performance

Figures 5.3a to 5.3h show the measured gross throughput of the cluster. The number of spouts or bolts is held constant while the other is scaled. Depicted is the total number of spouts or bolts, respectively.

We look at the benchmarks one by one:

- Byte streaming: A single thread is capable of sending 2.69 Gib/s, the spout being CPU-bound in the KryoCpp serializer. Adding more spouts shows near-linear scaling up to the point of maximum network throughput. Deserializing is even more efficient with a single bolt receiving 4.88 Gib/s. Again throughput is limited by the network when scaling up the number of bolts on a 10 Gib/s link and a single bolt on a 1 Gib/s link.

  The dual-threaded spouts show performance equivalent to two of their single threaded counterparts. This is unsurprising when we consider the implementation: each thread owns a private *TaskTools* instance. Absorb and emit in a multi-threaded task function identically to running multiple single-threaded tasks. The added value of a multi-threaded task is that a client can control the interaction of the threads, e.g. communication via a shared-memory model.

  Considering these results we argue that if a task is embarrassingly parallel there is no gain in threading a task. However, if tuples must be processed in a single instance, threading a task may effectively facilitate throughput bottlenecks.

  The slight performance variations seen at 10 Gib/s are due to test bed inconsistencies described in section 5.2.1.

- Distributing word count: A single spout is capable of emitting at 2.64 Gib/s and scales near-linearly until it is network-bound, fortifying our previous byte streaming results. As expected bolts are CPU-bound, a single bolt processing 0.7 Gib/s. With linear scaling at least 6 bolts are necessary to saturate two spouts at 3.67 Gib/s, or 14 bolts for saturating 4 spouts at 9.22 Gib/s. Since the scaling is not precisely linear, 16 bolts are necessary to reach the hardware limitations of the d820 nodes.
- Aggregating word count: As in the distributed word count, a single bolt processes 0.7 Gib/s. On our quad-core d710 NaaS box we are limited to running 4 bolts at 2.71 Gib/s. The NaaS box is CPU-bound, demonstrating the limitation of the NaaS concept. Nevertheless the speedup over the standard topology is 2.94. The spouts show throughput equal to the distributed word count benchmark, verifying the result.

  On the 32 core d820 nodes it is possible to scale to more bolts until the maximum network throughput is reached.

  In figures 5.3g and 5.3h we have included the d820 results from 5.3e and 5.3f again as a reference.
- Local data transfer: The spouts scale linearly up to 9.15 Gib/s on d710 nodes, at which point we are CPU-bound. As a single bolt already achieves very high throughput, the throughput of bolts is limited by that of the spouts. Considering that the 8 tasks are all running on the same CPU and the d820 transfers 15 Gib/s with 8 spouts, we speculate that a d710 node could emit tuples at a rate up to 15 Gib/s if given the networking equipment. Based on our results we argue that our NaaStorm implementation has headroom for more throughput.

In all benchmarks we see a considerable speedup when comparing the NaaS topology to the standard topology. Only when the NaaS box is CPU-bound does it not saturate the network links, but even here we observe a substantial speedup. At the same time we observe that a more powerful NaaS box enables scaling until we are network-bound. If a single NaaS box does not satisfy the network capacity, multiple NaaS boxes could again push the limits.

We argue that the best case of NaaS is full utilization of the available network capacity, the worst case is equal to the standard topology.

5.2 Discussion

5.2.1 Problems Encountered

- Test-bed problems:
  When starting out the project with evaluating ZeroMQ performance, we had issues with the NIC driver. Throughput would fluctuate for no apparent reason in both Iperf and ZeroMQ. After a update to the system, NIC performance was as expected. However, GCC now refused to link binaries. We could not find a solution to the problem and Clang did not have this issue, so development continued with Clang.

  Development of the concurrent lock-free queue necessitated use of compiler intrinsics, making code compiler-specific. Clang is a relatively new project and GCC currently has much better documentation of their intrinsics, hence the queue was developed with GCC.

  Finally, time slot constraints made it necessary to move test beds and all further development was done on Emulab.

  On Emulab we experienced throughput inconsistencies on d820 hardware. Symptoms included low throughput with high variance between benchmark runs and high variance between directions (node0 $\rightarrow$ node1 different from node1 $\rightarrow$ node0). Our solution was to repeat the benchmark runs until results were in an expected range. At the time of this writing (8 September 2012) the d820 nodes have just recently been announced and there remain further issues such as not all NICs being connected but not being recognized as non-functional by the Emulab link test.

- Storm documentation:
  As mentioned in the introduction chapter, Storm is a relatively young project. The first public release was on 17 September 2011. Documentation on Storm's concepts and Java and Thrift APIs existed and were useful for building our project. However, only recently (April 2012) has documentation of Storm's internal design and structure started to appear. At the time of this writing (18 June 2012) three of six parts listed in the index are pending.

  Much of our own knowledge on Storm had to be acquired by reading Clojure code, a language we were initially completely unfamiliar with. Thus assumptions were made on how various components work and interact. Unsurprisingly, many assumptions made were wrong and preliminary designs had to be revised. Towards the end of the project, more documentation surfaced and our overview improved as we spent time reading code and familiarizing ourselves with Clojure.

- Clojure type inference and Java:
  Clojure is a non-pure functional language designed to run on the JVM. Unlike other functional languages it supports *structs*, which make it possible to use and implement Java classes and functions by passing the struct as the first parameter. Sadly, the reverse case is not strictly enforced. While types of struct fields and function arguments <u>may</u> be defined, like in other functional languages type inference often makes this unnecessary.

  The Java *clojure.lang.rt* package enables the calling of Clojure functions from Java. The static return value of the package's functions is of type *Object*. While this is sufficient to call other Clojure functions, it does not enable further processing in Java itself. Thus the dynamic type must be manually worked out and the object cast to that type. This process greatly slowed us down when implementing the NaaStorm initializer.

- Storm message format:
  Knowing the Storm message format is essential to us because we must access the tuples and header information stored in the message's fields in the NaaStorm worker itself. This is presumably the last critical detail of Storm missing to complete the implementation of NaaStorm.

  The current status is that we do know the underlying communication layer, which is ZeroMQ and can be directly accessed from C++. We know that fields are serialized with Kryo and Storm's custom field serializer, which have both been implemented in C++. What remains to be done is to ascertain all information which is contained in fields and in which order it is serialized. This has so far eluded us, as the serialization is seemingly done in several layers within Storm and we are still not completely comfortable with Clojure as a language.

5.2.2 Limitations

- Storm integration:
  Due to time constraints we did not achieve integration of NaaStorm into Storm. The final hurdle to overcome is understanding and implementing the inter-task message format of Storm. However, NaaStorm can be used as a standalone application separate from Storm.

- Physical location of tasks in Storm cluster:
  As of now we cannot control the assignment of a certain task to a certain cluster node. This ability is critical for the NaaS model, without it we cannot assign tasks to a NaaS box at a switch.

  The undertaking is ambitious, as it requires understanding the task scheduler in Nimbus and then modifying it. Time constraints and limited knowledge of Clojure have prevented us from implementing our own solution. On the bug tracker a discussion[54] has taken place about this very feature, because other Storm tenants have specific needs for it. Their reasons are:

  - Manual load distribution in the cluster, as some tasks can be more resource intensive than others. Storm currently only considers the number of tasks per worker when scheduling.
  - Locality of two or more tasks, as sometimes few tasks communicate more intensively with one another than with other tasks. When assigned to the same worker tasks communicate via the ZeroMQ inproc protocol.
  - Software licensing limitations.

  The first version of Storm to be released with pluggable scheduler support is 0.8.0. A release candidate is estimated for July 2012.

- Task interface limitations:
  Our task interface was designed to be powerful while still being usable and not overly complex to implement. As such we made several trade-offs, which we discussed in section 3.4.1.

- Dynamic task loading:
  Tasks must be statically compiled into the NaaStorm worker, as this is sufficient for our purposes. The design could be extended to use the polymorphism of the task interface to dynamically load new task classes. [55] shows an example of how this can be accomplished in C++. Note that the tuple type used by dynamically loaded task must conform to the types the task interface is compiled with.

- Limited scalability with respect to number of target nodes:
  In section 3.4.2 we describe how and why we chose to give each outgoing port its own thread. The assumption that the number of destinations is small is easily refuted: spouts in large clusters could send messages to all bolts, possibly making the number destinations reach into the hundreds. The alternative design decision described would have been the better choice for large clusters, as the memory consumption for each new destination is smaller and adding new destinations does not impact the system's process scheduler.

- KryoCpp extensibility:
  KryoCpp is extensible only by implementing serializers as function template specializations. The design considerations, limitations and a possible future extension are discussed in detail in section 4.5.

- Reliability:
  As mentioned in section 3.3.2, to bound our project's scale we have neither implemented Storm's nor a custom reliability framework. If tuples are lost in transit and this leads to a failure or error, there is no provided recovery process.

Chapter 6

Conclusion

Limited network throughput between nodes is a bane for bandwidth-intensive cluster applications. The Network-as-a-Service architecture empowers tenants to tap into the network backbone and exploit the tree-hierarchy to their benefit. The abstraction of the physical topology of a cloud services is transparentized and control over task locality handed to the tenant.

NaaStorm has given Network-as-a-Service a face. We have demonstrated the potential of using the NaaS model in cluster applications and fulfilled our goal of saturating a 10 Gib/s network link in multiple benchmarks with headroom for even faster connections on current hardware. We have shown that even in the case of the NaaS box not being network-bound the larger bandwidth resources in our model are of value.

Along the way we have produced a C++ implementation of Kryo, demonstrating that the serialization format is not constrained to Java and that the already fast throughput can be further increased. In our evaluation of the ZeroMQ messaging library we have observed that the overhead over TCP/IP is minimal. The library has been a powerful asset, relieving us from designing an extensive communication framework ourselves.

Our implementation's high performance comes by way of an efficient design and careful engineering. Finical avoidance of memory copies throughout the program stack, cache locality and even reducing the number of pointer indirections have yielded performance gains. Choosing C++ over higher-level languages like Java enabled us to inspect the generated machine code and utilize advanced low-level profiling tools. However, NaaStorm is a proof-of-concept framework and as such lacks some practicability aspects such as task scheduling, dynamic task loading and reliability guarantees.

During the course of our project the goal of extending Twitter Storm to the NaaS model was exchanged in favor of a stand-alone solution. Lack of documentation and difficulties with the Clojure language hindered rapid progress. In the end the integration of our work had to be halted and work continued without Storm.

Although our efforts have yielded fruits, there remain additional, unanswered questions. How much additional effort is required to utilize NaaS in applications? How does NaaS impact other data center applications? Where are the limits of NaaS performance scalability in a larger, data center-sized setting? Integration with an existing distributed computation framework, porting real-world applications to the NaaS concept and large-scale tests would lead to more answers.

In retrospective, we would go about things differently. Being specific about as many design goals as possible early on helps to get a quick start. While unraveling the project as we went certainly was interesting, avoiding late changes to the project is very desirable. Yet we do understand that research is often open-ended and the laying of concrete milestones is not always practical. Writing background and documentation on-the-go is arduous, but it helps in reflecting design choices and feasibility in general. Stepping up our efforts there would also reduce the strain when finalizing the project. Lastly, evaluating the chosen base platform more diligently beforehand or designing for stand-alone operation from the start would save time.

In conclusion, we believe NaaStorm has proven the viability of Network-as-a-Service and laid groundwork for further research on the concept of in-network processing. The valuable lessons learned in the research process will carry us on in our future studies.

Bibliography

[1] A. Java, X. Song, T. Finin, and B. Tseng, "Why we twitter: understanding microblogging usage and communities," in *Proceedings of the 9th WebKDD and 1st SNA-KDD 2007 workshop on Web mining and social network analysis*, WebKDD/SNA-KDD '07, (New York, NY, USA), p. 56–65, ACM, 2007.

[2] "Twitter blog: 100 million voices." http://blog.twitter.com/2011/09/one-hundred-million-voices.html. Accessed: 8 Feb 2012.

[3] "Twitter blog: 200 million tweets per day." http://blog.twitter.com/2011/06/200-million-tweets-per-day.html. Accessed: 8 Feb 2012.

[4] J. Bollen, H. Mao, and X. Zeng, "Twitter mood predicts the stock market," *Journal of Computational Science*, 2011.

[5] T. Sakaki, M. Okazaki, and Y. Matsuo, "Earthquake shakes twitter users: real-time event detection by social sensors," in *Proceedings of the 19th international conference on World wide web*, p. 851–860, 2010.

[6] T. Benson, A. Akella, and D. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th annual conference on Internet measurement*, p. 267–280, 2010.

[7] T. Benson, A. Anand, A. Akella, and M. Zhang, "Understanding data center traffic characteristics," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 1, p. 92–99, 2010.

[8] "Cisco Systems — Data Center Design — IP Network Infrastructure." http://www.cisco.com/en/US/docs/solutions/Enterprise/Data_Center/DC_3_0/DC-3_0_IPInfra.html, November 2011. Accessed: 18 Jan 2012.

[9] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *ACM SIGCOMM Computer Communication Review*, vol. 38, p. 63–74, 2008.

[10] M. Blumrich, D. Chen, P. Coteus, A. Gara, M. Giampapa, P. Heidelberger, S. Singh, B. Steinmacher-Burow, T. Takken, and P. Vranas, "Design and analysis of the BlueGene/L torus interconnection network," *IBM Research Report RC23025 (W0312-022)*, vol. 3, 2003.

[11] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, "Bcube: a high performance, server-centric network architecture for modular data centers," in *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, SIGCOMM '09, (New York, NY, USA), pp. 63–74, ACM, 2009.

[12] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: a scalable and flexible data center network," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 4, p. 51–62, 2009.

[13] J. Mudigonda, P. Yalagandula, M. Al-Fares, and J. C. Mogul, "SPAIN: design and algorithms for constructing large data-center ethernets from commodity switches," tech. rep., Tech. Rep. HPL-2009-241, HP Labs, 2009.

[14] A. Shieh, S. Kandula, and E. G. Sirer, "SideCar: building programmable datacenter networks without programmable switches," in *Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks*, p. 21, 2010.

[15] P. Costa, M. Migliavacca, P. Pietzuch, and A. L. Wolf, "Naas: network-as-a-service in the cloud," in *Proceedings of the 2nd USENIX conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, Hot-ICE'12, (Berkeley, CA, USA), pp. 1–1, USENIX Association, 2012.

[16] "Gibhub: Storm wiki." https://github.com/nathanmarz/storm/wiki. Accessed: 18 Jun 2012.

[17] E. Curry, "Message-oriented middleware," *Middleware for communications*, p. 1–28, 2004.

[18] G. Banavar, T. Chandra, R. Strom, and D. Sturman, "A case for message oriented middleware," *Distributed Computing*, p. 846–846, 1999.

[19] E. Curry, D. Chambers, and G. Lyons, "Extending message-oriented middleware using interception," in *Third International Workshop on Distributed Event-Based Systems*, p. 32, 2004.

[20] "ØMQ the intelligent transport layer." http://www.zeromq.org. Accessed: 6 Jun 2012.

[21] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, MA: Addison-Wesley, 1995.

[22] Intel, Santa Clara, CA, USA, *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference, A-M*, Sept. 2009.

[23] Intel, Santa Clara, CA, USA, *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2B: Instruction Set Reference, N-Z*, Sept. 2009.

[24] "Stackoverflow: 0mq: How to use zeromq in a threadsafe manner?." http://stackoverflow.com/a/5842249. Accessed: 16 Jun 2012.

[25] "Zeromq: The guide - a request-reply broker." http://zguide.zeromq.org/page:all#A-Request-Reply-Broker. Accessed: 18 Jun 2012.

[26] A. Sumaray and S. Kami Makki, "A comparison of data serialization formats for optimal efficiency on a mobile plattform," (Kuala Lumpur, Malaysia), ACM, Feb. 2012.

[27] M. Hericko, M. B. Juric, I. Rozman, S. Beloglavec, and A. Zivkovic, "Object serialization analysis and comparison in java and. net," *ACM Sigplan Notices*, vol. 38, no. 8, p. 44–54, 2003.

[28] "Java object serialization specification." http://docs.oracle.com/javase/7/docs/platform/serialization/spec/serialTOC.html. Accessed: 7 Jun 2012.

[29] L. Opyrchal and A. Prakash, "Efficient object serialization in java," in *Electronic Commerce and Web-based Applications/Middleware, 1999. Proceedings. 19th IEEE International Conference on Distributed Computing Systems Workshops on*, p. 96–101, 1999.

[30] K. Kono and T. Masuda, "Efficient RMI: dynamic specialization of object serialization," in *Distributed Computing Systems, 2000. Proceedings. 20th International Conference on*, p. 308–315, 2000.

[31] C. Nester, M. Philippsen, and B. Haumacher, "A more efficient RMI for java," in *Proceedings of the ACM 1999 conference on Java Grande*, p. 152–159, 1999.

[32] M. Wegiel and C. Krintz, "Cross-Language, Type-Safe and transparent object sharing for Co-Located managed runtimes," ACM, 2010.

[33] "kryo - fast, efficient java serialization and cloning." http://code.google.com/p/kryo/. Accessed: 6 Jun 2012.

[34] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, p. 107–113, 2008.

[35] Z. Xiao, H. Chen, and B. Zang, "A hierarchical approach to maximizing MapReduce efficiency," pp. 167–168, IEEE, Oct. 2011.

[36] Y. Luo, Z. Guo, Y. Sun, B. Plale, J. Qiu, and W. Li, "A hierarchical framework for cross-domain MapReduce execution," in *Proceedings of the second international workshop on Emerging computational methods for the life sciences*, p. 15–22, 2011.

[37] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava, "Building a high-level dataflow system on top of map-reduce: the pig experience," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, p. 1414–1425, 2009.

[38] R. Lämmel, "Google's mapreduce programming model–revisited," *Science of Computer Programming*, vol. 70, no. 1, p. 1–30, 2008.

[39] B. He, W. Fang, Q. Luo, N. Govindaraju, and T. Wang, "Mars: a MapReduce framework on graphics processors," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, p. 260–269, 2008.

[40] "Apache thrift." http://thrift.apache.org. Accessed: 14 Jun 2012.

[41] "Linkedblockingqueue (java platform se 6)." http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/LinkedBlockingQueue.html. Accessed: 16 Jun 2012.

[42] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," in *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '06, (New York, NY, USA), pp. 339–350, ACM, 2006.

[43] "Cisco catalyst 6500 supervisor engine 32 programmable intelligent services accelerator." http://www.cisco.com/en/US/prod/collateral/modules/ps2797/ps7209/prod_bulletin0900aecd805a0e30.pdf. Accessed: 11 Jun 2012.

[44] "HP thread management services zl module." http://h18000.www1.hp.com/products/ quickspecs/13376_div/13376_div.PDF. Accessed: 11 Jun 2012.

[45] "Loadable kernel module programming and system call interception | linux journal." http://www.linuxjournal.com/article/4378. Accessed: 11 Jun 2012.

[46] "What is the LD_PRELOAD trick? - stack overflow." http://stackoverflow.com/questions/426230/what-is-the-ld-preload-trick/426260. Accessed: 11 Jun 2012.

[47] "The maximum length of arguments for a new process." http://www.in-ulm.de/ mascheck/various/argmax. Accessed: 18 Jun 2012.

[48] "Boost C++ libraries." http://www.boost.org. Accessed: 17 Jun 2012.

[49] "String (Java 2 platform SE 5.0)." http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/String.html. Accessed: 7 Jun 2012.

[50] H. Sutter, "Why not specialize function templates?," *C/C++ Users J.*, vol. 19, pp. 65–68, July 2001.

[51] B. Meyer, *Object-oriented software construction (2nd ed.)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1997.

[52] "Status of experimental C++11 support in GCC 4.7." http://gcc.gnu.org/gcc-4.7/cxx0x_status.html. Accessed: 28 Aug 2012.

[53] "ØMQ lightweight messaging kernel (v0.1)." http://www.zeromq.org/whitepapers:design-v01. Accessed: 2 Sep 2012.

[54] "Issue #164 pluggable scheduler - nathanmarz/storm - github." https://github.com/nathanmarz/storm/issues/164. Accessed: 18 Jun 2012.

[55] "Dynamic class loading for C++ on linux | linux journal." http://www.linuxjournal.com/article/4378. Accessed: 17 Jul 2012.