Failure detectors in Erlang

Kim Hammar

June 24, 2017

1 Introduction

This report covers the work done in a distributed system assignment implemented in the Erlang programming language. The assignment involved experiments with failure detectors and particularly Erlang's built-in failure detector support.

2 Main problems and solutions

- *Implementation*
  The implementation for this task was very trivial and just involved two processes pinging each other and having one process monitoring the other process with Erlang's built-in failure-detector.
- *Experiments*
  Experiments were conducted to investigate how the semantics of Erlang's failure detector works in practice.

2.1 Results

Local deployment Running the application in non-distributed mode on a single machine, the crash of the producer process was detected almost instantly with the message:

```
{producer,nonode@nohost} died; {badarith,....}
```

When running the two Erlang nodes locally and crashing the whole producer-node the consumer instantly detected the crash and output:

```
{producer,gold@limmen} died; noconnection
```

Distributed deployment In the distributed setting things started to get a bit more interesting.

When unplugging the Internet connection of the producer node for a few seconds the consumer didn't notice anything particular except a disruption in the pings and when the producer was re-plugged the consumer received ping messages with wrong sequence number.

Unplugging internet of the producer for a longer period of time revealed some good hints of how Erlang's failure detector works. It took $55s$ until the consumer detected that there was no connection to the producer and output:

```
** Node 'silver@192.168.1.91' not responding **
** Removing (timedout) connection **
```

Another observation was that even after the consumer detected the crash of the producer, when re-plugging the internet connection to the producer the consumer continued receiving ping messages just like before.

Semantics of Erlang So what does these error messages tell us about Erlang monitor semantics and guarantees?

Well first of all, an important observation here is that when the producer is explicitly crashed the consumer outputs the actual error which we can interpret as the consumer for sure knows that the producer has crashed because it explicitly received a exit-message with reason of crash from the producer. In the case where the network connection between the producer and consumer is aborted the consumer only outputs that the producer is "not responding" i.e we can interpret this as the consumer cannot tell if the producer is crashed or just slow in responding. This is the classical two generals problem which is inherent in any distributed system.

The Erlang documentation says as follows about the message delivery guarantees for Erlang:

> *A 'DOWN' message will be sent to the monitoring process if Item dies, if Item does not exist, or if the connection is lost to the node which Item resides on.*
>
> *"Delivery is guaranteed if nothing breaks" - and if something breaks, you will find out provided you've used link/1. I.e. you will get an EXIT signal not only if the linked process dies but also if the entire remote node crashes, or the network is broken, or if any of these happen before you do the link.*

So how does the Erlang monitor work internally? It uses heartbeats/pings between Erlang nodes that are connected to each other to detect which nodes it has a connection to and which nodes are not responding. Reading in the Erlang documentation it seems like the heartbeat between nodes that are connected occurs every 15s and after 4 missed heartbeats (60s) the node declares the other node as not responding.

3 Conclusions

If explicitly crashed, Erlang monitor will detect the crash almost instantly since it will receive and exit-signal from the crashed process, if there is a network partition however, it takes longer. Erlang monitors seems to use a timeout of 60s as default, if there is not response within 60s the Erlang monitor will conclude that the node is not responding. According to the Erlang mailing-list the timeout can be configured with the flag `-kernel net_ticktime T`.

Erlang's message passing is built on top of TCP and the semantics can be regarded as FIFO reliable delivery. I.e if both sender and receiver is correct and assuming network partitions eventually heal, messages from A to B will eventually be delivered by B (It might be that messages sent during the partition gets lost and not resent later, I could not find any information on this in the documentation). If network partitions does not heal or if some process is faulty the FIFO guarantee still holds but it is not guaranteed that messages are eventually delivered.

Seminar 5
Chordy: a distributed hash table

Kim Hammar

October 13, 2016

1 Introduction

This report covers the work done in an assigment on distributed systems. A basic variant of the Chord protocol was implemented in Erlang. Performance tests and a smaller analysis of consistent hashing, Chord and P2P networks was made.

2 Main problems and solutions

The main challenges encountered for this assignment were:

- *Implementation of a structured P2P network that is decentralized* Implementation is based solely upon message passing in Erlang. Implementation includes building a "ring" of nodes, maintaining ring-stability when nodes join and leave, distributed storage and achieving fault tolerance through replication and failure detection. Implementation have also been extended to optimize for routing-performance with finger-tables.
- *Examining how scalable the ring is and what bottlenecks to consider* Basic benchmarks were performed and analyzed.

2.1 Implementation

Nodes in the network are implemented as erlang processes that communicates solely though message passing. (Imperfect) failure detectors are constructed with erlang monitors. Distribution is trivial with erlang as the underlying platform.

2.2 Building a ring

To achieve a completely decentralized P2P network that uses the Chord protocol, nodes are structured in a *ring*, an overlay network where each node (peer) has the same role. To construct a conceptual ring of nodes the essential requirement is that each node knows about its *successor*, which is the next node in the ring (clockwise). Nodes are assigned $m-$bit identifiers (keys) and are placed in the ring according to their key. To join an existing ring you need to know at least one node that is a member of the ring. It is common to use some kind of bootstrap-server that give out keys to new nodes as well as the location of an existing node in the ring that the new node can contact in order to join the ring.

Nodes in the ring are responsible for keys in the range $(predecessor_{key}, key]$, conceptually the range of keys that a node in the ring is responsible for is the "hole" in the ring between itself and its predecessor. A desired property of the ring is that nodes are evenly distributed, generally a hash-function is used to achieve this. In a ring where nodes are evenly distributed the notion of *responsibility* is important, since that it was makes the location of keys *consistent*. In a perfectly balanced ring of N nodes, only $\frac{1}{N}$ keys are moved when a node join or leave the ring.

2.3 Stabilization

Chord uses a stabilization protocol that is ran periodically to check if the ring is stable or if some re-ordering is neccessary. The purpose of the periodic stabilization is to keep the ring stable when new nodes join, it does so by letting each node communicate with its successor. The stabilization protocol does not give any semantics for dealing with failures of other nodes.

2.4 Failures and Replication

Dealing with failures:
The correctness of the Chord protocol relies on the fact that each node knows its successor. To achieve fault-tolerance it is essential that each node in the ring can find a new successor in case of a crashed successor. The recommended way of achieving this in Chord is to let each node maintain a successor-list of size r. Each node in the ring should be equipped with two failure detectors, one for its successor and one for its predecessor. If a successor crashes the predecessor will eventually notice it and update its successor pointer to the next node in the successor-list. If a predecessor crashes the successor should clear its predecessor pointer so that it later can

add a new predecessor. With a list of size r and if node-crashes happens with probability p then the ring can continue functioning with $1 - r^p$ probability. Implementation wise the successor list is maintained by a periodic procedure where each node queries its successor for its successor, and then the next successor and so forth.

Replication:
By following the guidelines above one can achieve a P2P network that is fault tolerant in the sense that it can rearrange itself to continue functioning despite failing nodes. However, just being able to maintain the ring is not enough if you want to achieve fault tolerance, you also want to be able to preserve the data in case of failures, this is achieved through *replication*. At a minimum every node should know about its successor and can replicate data there, but a general rule of thumb is to have a replica-degree of atleast 3. Where to replicate depends on the purpose of the replication (performance, availability, load-balancing etc.), the simplest replication schema in Chord for high availability is to replicate at nodes in the successor list. Replication at successors is convenient since if a node crashes the node that will take over responsibility of the keys of the crashed node is the successor, thus successor replication minimizes data-transfer. Ofcourse replication is not as easy as just sending replicas to each node in the successor list, you need to deal with consistency, replication coherency and transfer of replicas when nodes join/crash just to name a few things.

2.5 Routing performance

Routing of requests in a Chord ring is a $\mathcal{O}(N)$ operation. However, the time to route requests in Chord can be reduced in most cases by maintaining routing information at each node. In Chord-terms this routing information is called *finger-tables*, which works like a routing table with fingers to $\log N$ nodes in the ring. If finger-tables are used then with high probability a routing request will only be $\mathcal{O}(\log N)$. Finger-tables where implemented in an extended module `node5.erl` and benchmarks illustrating the effectiveness of finger-tables is presented in the sequent section.

3 Evaluation

3.1 Setup

Test scenario:
Perform 1000 sequential lookup for keys in a chord ring. Varying parameters between the tests are:

- **Number of nodes in the ring**
  Five cases where examined: $1, 25, 50, 100$ and 200 nodes. The keys are evenly distributed among the nodes: $\frac{1000}{nodes}$ keys per node.
- **Routing technique in the ring**
  Two cases:
  - *Standard successor-routing*, nodes don't maintain any routing information. If a node gets a lookup-request for a key that it is not responsible for it will forward the request to its successor.
  - *Finger-table routing.* Each node maintains a finger table of size m where m is the bit-length of identifiers for keys. In this test $m = 30$.
- **Latency**
  The ring of nodes was hosted on a single machine for the test, communication latency of 1ms for routing of messages was simulated with: `timer:sleep(1)`.

Nodes	**Finger-table routing, time(s)**	**Successor-routing, time(s)**
1	0.001	0.001
25	4.40	24.00
50	5.40	252
100	6.32	347
200	7.33	199.1

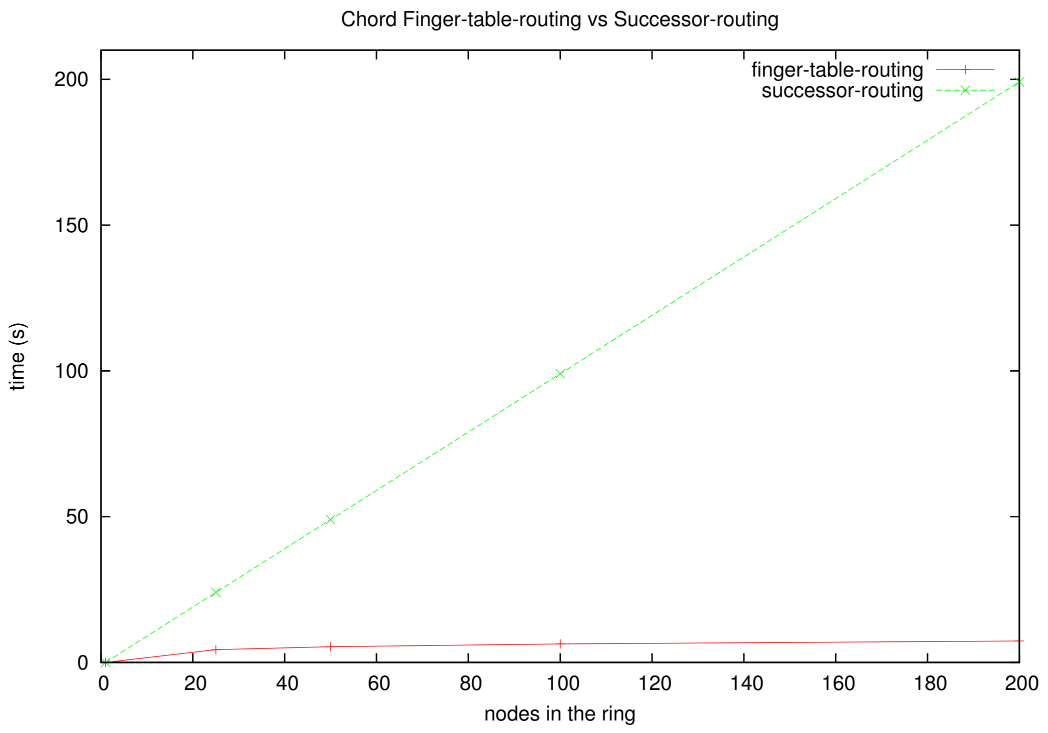


Figure 1: Test benchmarks

3.2 Analysis

The test results illustrates the ideal scenario for using finger-tables, when all fingers are correct, lookup time is reduced drastically. The finger-table routing achieves a $\mathcal{O}(\log N)$ growth when the ring gets larger compared to successor-routing which follows a linear expansion in routing latency.

In other settings finger-tables will probably not be as effective as shown in this test. A ring that has a high churn-rate will increase the probability of incorrect finger-tables which will reduce the efficiency of finger-tables. Also, something that is not shown in the test is that maintaining finger tables entails an overhead and increased usage of bandwidth since nodes need to exchange more messages between each other.

4 Conclusions

A decentralized p2p system is an attractive choice for systems that requires high availability and scalability. Chord is a protocol for building a self-organizing distributed hash-table. The protocol is designed to handle nodes joining/leaving in a dynamic environment. The core concept of the protocol is relatively simple and easy comprehensible. Challenges arise when optimizing the implementation for performance and fault-tolerance. A decentralized system can be a real challenge to debug and reason about, especially in combination with other difficult areas such as replication, consistency, routing etc.

Seminar 4
Groupy: a group membership service

Kim Hammar

October 6, 2016

1 Introduction

This report covers the work done in an assignment on distributed systems. The given task was to implement a group membership service that provides atomic multicast in Erlang. The purpose of the assignment was to learn about the challenges involved and the theory behind group communication in a distributed system.

2 Main problems and solutions

The main challenges encountered for this assignment were:

- *Coordination of state among multiple application processes on different nodes.*
  This is achieved by requiring nodes that want to update the state to first send a request to the multicast layer which later will amount to a multicast of the request to all members of the group. When the group members receive the request they can update their state accordingly.
- *The group service should be reliable under certain assumptions.*
  The implementation should be reliable under the following assumptions:
  Assumption 1-$\mathcal{A}$. *Messages are reliably delivered.*
  Assumption 1-$\mathcal{B}$. *Erlang monitors behaves as perfect failure detectors*
  Assumption 1-$\mathcal{C}$. *The Erlang system provides FIFO ordering of messages between two processes*

  That makes it slightly unrealistic, but even with those assumptions there is a challenge to make the service reliable, it must tolerate process crashes.

2.1 Coordination of state and views

Application processes will keep track of some state that should be synchronized. Additionally, the application processes should have a *consistent view* of the current group membership. The views are *delivered* to the application process from the group membership service.

In this assignment the approach of a **Coordinator** is used. One arbitrary process in the group will be the designated coordinator or *leader*. The rest of the processes in the group are *slaves*. All messages that are directed towards the group will be routed by the slaves to the leader. The leader will then multicast the message to every member in the group.

Given the assumptions 1-$\mathcal{A}$ and 1-$\mathcal{C}$, the only thing necessary to achieve atomic multicast, view-synchrony and total order of messages in the group is to add sequence numbers to messages and use erlangs built-in selective receive to pick out messages in the correct order from the mailbox. The processes should then receive multicasted messages at the same logical time.

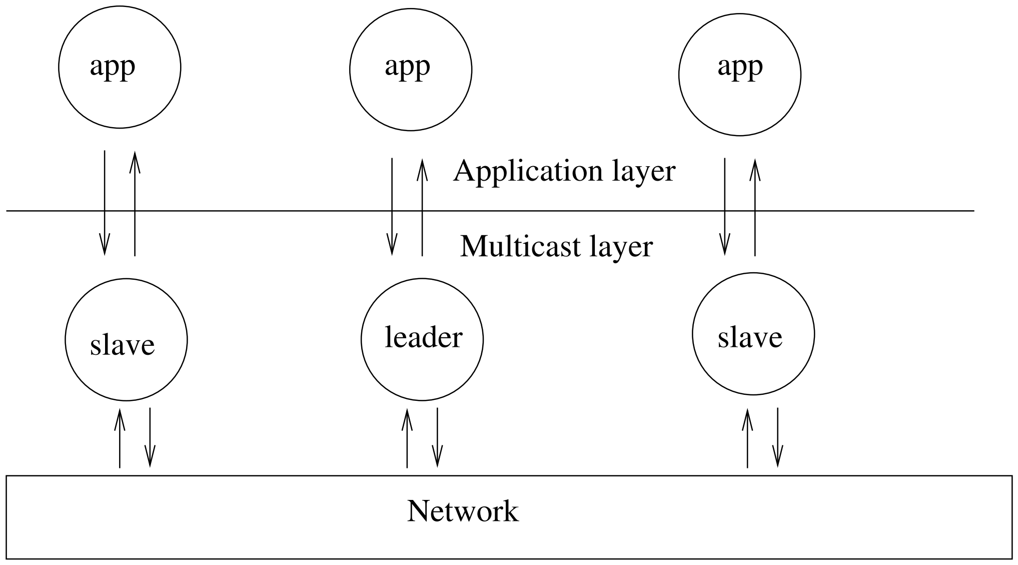


Figure 1: Architecture

2.2 Reliability

The mechanisms above will not handle any type of failures. Since there is no assumption that guarantees that nodes wont crash it is neccessary to implement certain reliability mechanisms.

With the assumption 1-$\mathcal{B}$, we equip every slave with a failure detector for the group leader. For simplicity we dont bother to add failure detectors to slaves since it does'nt matter for the application-processes if the view of the group membership contains dead slaves. The proof-of-concept for this implementation is that the view will be consistent across all nodes.

If the leader process crashes, all slaves will detect this through their failure detectors. The slaves are then to elect a new leader, to do this we utilize that every slave already possess a consistent view of the group. A simple rule that the first slave in the list of slaves should be elected the new leader is enough to reach consensus on who to elect (**Note:** there is a possibility that a crashed node is elected leader, but due to the failure detectors this will be detected immediately and a second election is initiated).

There is a special case when the atomicity of the multicast can be violated despite having failure detectors: If the leader crashes during a broadcast it might happen that it have sent messages to some nodes but not all.

To accomodate this we add a mechanism where a newly elected leader will always resend the latest message, that will make sure that every member of the group receives it. Since some nodes might receive duplicates it's also neccessary that the slaves are able to filter out duplicate messages.

2.3 Analysis

This implementation guarantees synchronized views for a group of processes. The implemention does handle failures without adventuring the synchronization.

Total order: Given that all broadcasts go through the designated leader process and that erlang provides FIFO ordering between two processes and that our implementation uses sequence numbers and selective-receive, all *correct* processes of the group will see messages in the same order (dictated by the leader) as long as messages are not lost.

However, as mentioned, this implementation relies on unrealistic assumptions. In a distributed system you cannot guarantee that omission failure wont occur. For example the network might fail, which makes assumption 1-$\mathcal{A}$ a fallacy. If we cannot assume 1-$\mathcal{A}$ then it follows as a consequence that we cannot assume 1-$\mathcal{B}$ either. The only assumption that the Erlang system actually guarantees is 1-$\mathcal{C}$.

Problems that arises when we falsify assumptions 1-$\mathcal{A}$ and 1-$\mathcal{B}$:

- Omission failures might occur and thus we cannot assume that a simple broadcast will reach every process in the group.

- Since we dont have access to a perfect failure detector, workers might think the leader is dead when it is not.

3 What could possibly go wrong?

As stated in the assignment:

> *-our implementation does not work*

When we developed our solution we promised *atomic FIFO broadcast* but what if a message is lost and this situation happen:

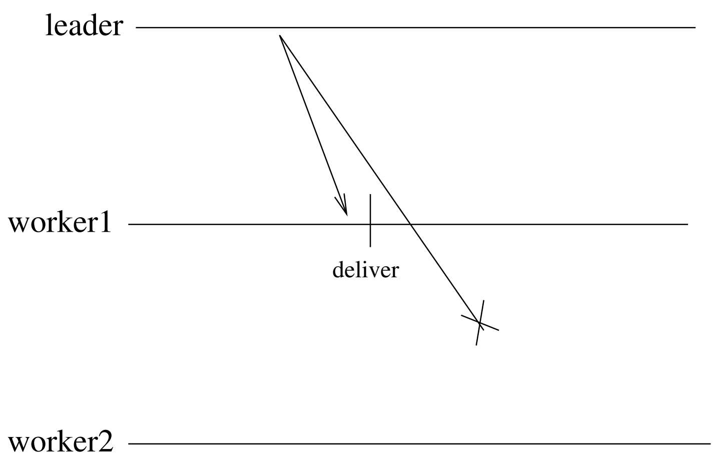


Figure 2: Message loss

This is something that can occur in a asynchronous system and with our current implementation it will cause the workers to come out of sync.

If we assume that network-partitions don't happen, it is possible to extend our current solution to give correct behaviour in *most cases* (atomic multicast is reducible to consensus), despite message-loss and process-failure.

A 2-phase-commit protocol is introduced to deal with message loss and the accuracy requirement of the failure detector is relaxed and only require liveness and safety (processes might declare other correct processes as suspected), this way the algorithm will always make progress despite failures.

3.1 Solution

Problem 1. The network is unreliable. $\longrightarrow$ each message should be acknowledged.

Problem 2. Acknowledgements can be lost. $\longrightarrow$ resend a few times, if still no acknowledgement is received, declared the process to be faulty and keep progressing.

In a module `gms4` the group membership service have been extended as follows:

i Slave processes in the group keeps a holdback-queue of uncommitted messages/views.

ii When a slave process receives a multicasted message from the leader, it does'nt deliver it to the application layer, instead it adds it to the holdback-queue and sends back an acknowledgement to the leader.

iii For each message/view that the leader multicasts it will initiate a *two-phase-commit* that works as follows:

   (a) The leader multicast the message/view to each slave

   (b) The leader collect acknowledgements from the slaves, if a timeout occurs before all acknowledgements have been collected, the leader resends the original message to the slaves that did'nt respond and then collects acknowledgements again. This is done a number of times until, either all slaves have acknowledged the message, or until the message have been resend more times than a limit-value.

   (c) If a slave doesn't acknowledge the message after multiple resends, the leader will discard the process as faulty, kill it and then continue progressing. **Note:** in a real group membership service that follows view-synchrony, the leader would install a new view without the dead process before it issues the commit, but to not do too many changes in `gms4` compared to `gms3`, dead processes are still kept in the view.

   (d) After acknowledgement from all correct processes have been received, the leader broadcasts a *commit-message* to all slaves that acknowledged the original message.

   (e) After multicasting the commit-message the leader repeats the acknowledgement-procedure described above until it have received acknowledgements for the commit-message by all slaves, or killed of non-responding ones.

(f) When a slave receives a commit-message tagged with a recent sequence-number it will extract the message from the holdback-queue of uncommitted messages and deliver it to the application layer, and then send an acknowledgement to the leader.

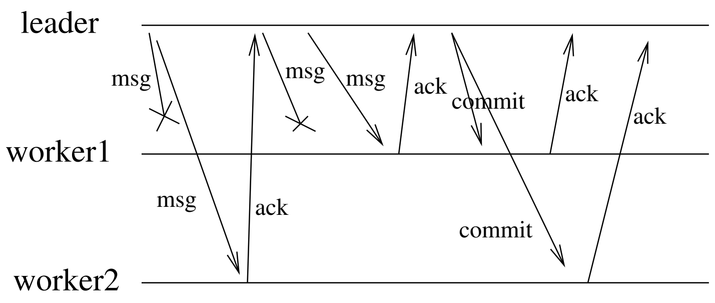


Figure 3: Distributed commit protocol to cope with message loss

3.1.1 Does it actually work?

- Yes, in a non-partitionable group.

Introducting a simulation of lost messages, similar to how process-crashes is simulated, show that the `gms4` module does handle message loss while `gms3` doesn't. With the `gms4` module, state-changes take a little bit longer than for the `gms3` module. If the maximum-amount of resends is set to 3, the protocol of `gms4` module will exchange $\mathcal{O}(6N)$ messages while the `gms3` protocol only exhanges $\mathcal{O}(N)$ messages for each multicast.

However, the `gms4` module will make sure that the state among the workers will always be consistent (non-responding workers are killed of). Some experiments show that setting the probability of message-loss as high as 25% is no problem for `gms4` using 3 resends as maximum, while `gms3` will loose synchronization immedieatly.

On the other hand, the `gms4` will be more likely to suspect correct processes to be faulty than the `gms3` module. In a real-world scenario the timeout-value that trigger resends and eventually killing processes should be decided by some machine learning algorithm that learns the average communication latency in the network.

3.2 Once more, what could possibly go wrong?

The third reason why things do not work is that we could have a situation where one incorrect node delivers a message that will not be delivered by any correct node. This could happen even if we had reliable send operations and *perfect failure detectors. How could this happen and how likely is it that it does? What would a solution look like?*

This is the scenario when the leader starts a broadcast and crashes before the broadcast is finnished such that only a subset of the processes in the group receives the message and then delivers it to the application layer, **and** then the group processes that received the message also crashes, see figure 4.

A solution to this problem is a solution that give *uniform agreement*, this is not possible in an asynchronous system, atleast one of the processes that received the commit-message needs to survive, otherwise the commit-information is lost even if the surviving processes reform the group and elect a new leader.

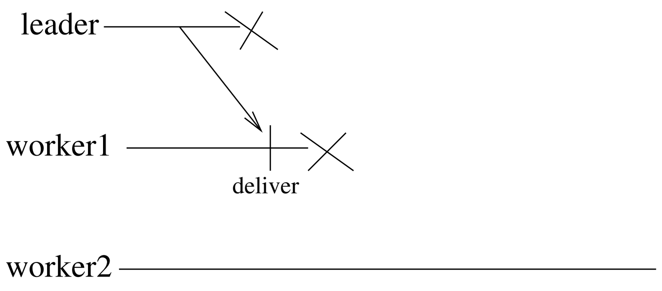


Figure 4: Non-uniform agreement scenario

Seminar 3
Loggy: a logical time logger

Kim Hammar

September 30, 2016

1 Introduction

This report covers the work done in an assignment on distributed systems. The given task was to implement a logical time logger in Erlang with the intent of learning about logical time with a practical example.

2 Main problems and solutions

The main challenges encountered for this assignment were:

- *Analyzing how we can see if the event-log from a set of workers are in order or not*
  A obvious difference of distributed computing compared to local computing is the latency of the sending and recieving of messages. Latency can cause messages to arrive out of order. In this assignment latency were simulated with random timeouts.
- *Implement logical time (lamport time)*
  Logical time and lamport clocks have been implemented to be able to apply a partial ordering of the events according to the happened-before relation. The assignment requires ordering of the events during execution, which introduces further challenges in determining the order of events *before* all events have been logged.
- *Implement vector clocks*
  Vector clocks is an extension to the concept of lamport time. Vector clocks solves some of the shortcomings of lamport clocks, but at the expense of a more memory expensive timestamp that grows proportional to the number of workers.

2.1 Loggy

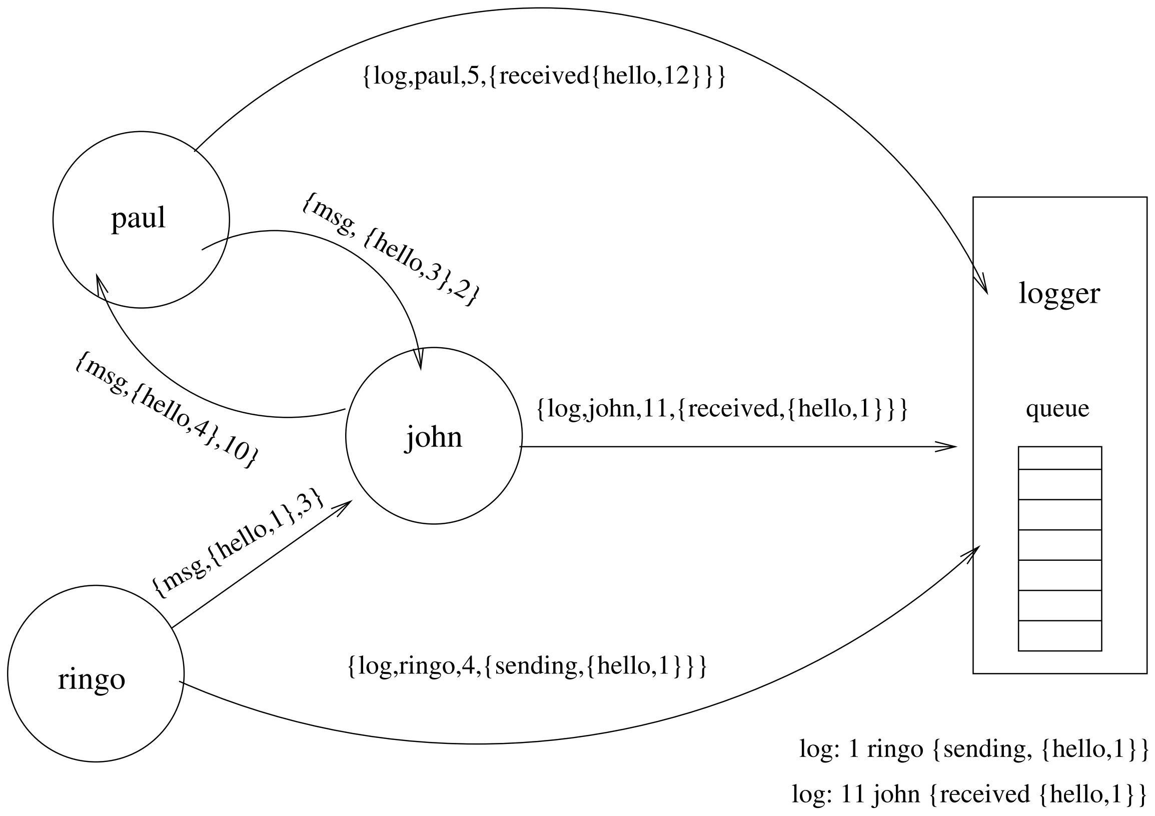

{log,paul,5,{received{hello,12}}}
paul
{msg, {hello,3},2}
logger
{msg,{hello,4},10}
john
{log,john,11,{received,{hello,1}}}
queue
{msg,{hello,1},3}
ringo
{log,ringo,4,{sending,{hello,1}}}
log: 1 ringo {sending, {hello,1}}
log: 11 john {received {hello,1}}

Figure 1: Loggy

2.2 Analyzing the log

Before implementing logical time and adding lamport timestamps to log messages, a test run (with simulated latency) could give the following log:

```
log: na ringo {received,{hello,57}}
log: na john {sending,{hello,57}}
log: na john {received,{hello,77}}
log: na ringo {sending,{hello,77}}
```

The log above is out of order. For example, according to the log, `{received,{hello,57}}` happened before `{sending,{hello,57}}`, which is not right. This unorder can happen due to communication latency since the log messages are sent by different workers, `{received,{hello,57}}` is logged by `ringo` and `{sending,{hello,57}}` is logged by `john`. There is however nothing from the log that implies that log-messages from the same worker are out of order, which is to be expected considering that the Erlang system gives a FIFO order of message delivery between two processes.

If we add lamport timestamps to the log messages but still allow the logger to print the log messages in the order it received them we could get the following log after a test run:

```
log: 2 ringo {received,{hello,57}}
log: 1 john {sending,{hello,57}}
log: 4 john {received,{hello,77}}
log: 3 ringo {sending,{hello,77}}
log: 4 ringo {received,{hello,68}}
log: 1 paul {sending,{hello,68}}
```

The lamport timestamps induce a partial ordering of events according to the happened before ($\rightarrow$) relation. Events that are causally related can be ordered totally but events that are concurrent can not. For example, `log: 1 john {sending,{hello,57}}` and `log: 1 paul {sending,{hello,68}}` happened concurrently and can only be ordered if we introduce some tie-breaker function.

2.3 Implementing logical time

The tricky part is for the logger to decide, by looking at timestamps and its own logical clock, when it is *safe* to log a message, we define a message to be *safe* to log if it does'nt break the partial ordering of timestamps. The logger should print all messages in order of their timestamps, this can not give a incorrect order according to the $\rightarrow$ relation, but it also means that there will exist multiple "correct" orderings.

If the logger has a clock C that keeps track of timestamps $t_1...t_n$ from the last messages received from each of the workers $1...n$, the decision if it's *safe* to log a message with timestamp t_i can be implemented as follows: If $t_i \leq t_j \quad \forall j, j = 1, ..., n$ then it is safe to print the log message with timestamp t_i (under the assumption of FIFO delivery of messages between two processes that erlang provides). Sketch of a proof:

Proof: By contradiction.

Assume $t_i \leq t_j \quad \forall j, j = 1, ..., n$ and it is **not** safe to print the message with timestamp t_i, then there must exist a message k with timestamp $t_k < t_j$ that is sent by the same worker as the message with timestamp t_j and arrived out of order (message j arrived before message k). This contradicts the assumptions declared above. $\square$

- *What does the ordered log tell us?* - The log tells us the order in which the events happened according to the happened-before ($\rightarrow$) relation. It does not tell us the order in which the events occured according to physical time.
- *How large will the holdback queue be?* - The holdback queue can in theory continue to grow forever, if one worker decides to never send any message to the logger it will prevent the logger from updating its clock, which will cause the holdback queue to stack up when the logger receives messages from the other workers.

Something that I initially got wrong when implementing logical time that made messages to be logged out of order was that I did'nt process the holdback queue in the right order, this was solved by sorting the holdback queue after the timestamps before processing it and logging messages that are safe.

2.4 Implementing vector clocks

The vector clock implementation gives some benefit compared to the lamport clock in that it provides more information about the causal relationship between events, but it does so at the expense of being more memory expensive, the vector clock grows proportional to the number of workers.

Comparison between two test runs:

Two main observations when comparing the two clock implementations,

1. With the lamport clock the holdback queue grows alot bigger than with the vector clock, and when the logger is stopped there is generally a bunch of messages in the holdback queue that is flushed. In contrast with the vector clock, the log messages are printed quicker and it is rare that there are messages in the holdback queue that have not yet been printed when the logger stops.
2. The two different clocks gives different order in which messages are logged. Both orders preserves causal ordering according to the $\rightarrow$ relation.

For example, if we look at the two first log-entries from a testrun with both clocks we can see that the entries differ:

```
(lamport clock)

log: 1 paul {sending,{hello,68}}
log: 1 john {sending,{hello,57}}
```

```
(vector clock)

log: [{john,1}] john {sending,{hello,57}}
log: [{ringo,1},{john,1}] ringo {received,{hello,57}}
```

Because the lamport clock implementation is just a integer counter, the logger cannot safely print the `log: 2 ringo {received,{hello,57}}` message before it have received a message with lamport timestamp ≥ 2 from all of the workers. This is because as far as the logger know, the message that `ringo` received that triggered the log message `log: 2 ringo {received,{hello,57}}` could have been sent by any worker. This means that with the lamport clock the logger will always print all initial *send operations* ($timestamp = 1$), before any receive operation can be printed.

With the vector clock implementation, when the logger receives `log: [{ringo,1},{john,1}] ringo {received,{hello,57}}` it only has to wait before it has seen a message with timestamp ≥ 1 from `john` and `ringo`, it does **not** have to wait until it have received a message with timestamp ≥ 1 from *all* workers. This is because the extra information that the vector clock gives compared to a lamport clock allows the logger to compare the timestamps *for each* worker separately. The result is that the logger can print log messages faster and does not need to store as many messages in the hold-back queue.

Lamport clocks are not strongly consistent, for two arbitrary events i and j with associated timestamps t_i and t_j: $i \rightarrow j \implies t_i < t_j$, but $t_i < t_j \not\Rightarrow i \rightarrow j$. This makes it appear that events are causally related (different timestamps) even when they are not. For example say that we have four processes, two processes send messages between each other and the other two send messages between each other, when looking at the log it will seem like all of the 4 processes are strongly causally related while in reality they are only pairwise causaully related.

Vector clocks are strongly consistent and solves this problem. For any two workers p_i and p_j with vector clocks v_i and v_j, and for any event x that happened at p_i we know the following: $x \rightarrow y \implies v_i[i] < v_j[i]$, and $v_i[i] < v_j[i] \implies x \rightarrow y$. Which means that after execution we can analyze the log more completely to find out which events are causally related and which happened concurrently.

3 Conclusions

Ordering of events in a distributed system is not trivial, with that said it is quite amazing how much you can achieve with the idea of logial time and just maintaining a simple counter timestamp that is piggybacked to messages.

Developing applications that are distributed are very different from developing concurrent applications on the same machine. Many assertions that your program typically rely on in local computing is no longer true and you have to expect things to be more unreliable.

Namy: a distributed name server

Kim Hammar

July 2, 2017

1 Introduction

This report outlines the solution to a assignment on distributed systems, namely to implement a distributed name server that is similar to DNS. The purpose of the assignment was to get familiar with the principles of DNS and caching data in a tree structure.

2 Main problems and solutions

- *Implementation*
  Implementing the distributed name server was not a lot of work given the built-in primitives for distribution in Erlang and the given skeleton code. What required some caution was when implementing the cache and the recursive lookup which is not crystal clear at first sight.
- *Experimenting with the cache*
  The main undertaking for this assignment was experimenting with the cache properties, in particular modifying the TTL of the cache entries and measuring how this affects traffic and how this affects accuracy of lookups.

2.1 Implementation

The global Domain Name System keeps track of the domain-names on the Internet. It is probably the world's most distributed database and it follows a tree-structure. DNS maps domain names to IP-addresses. The implementation for this assignment is a miniature of DNS that keeps track of domain-names of hosts in the distributed Erlang application. It maps domain names to Erlang process-identifiers.

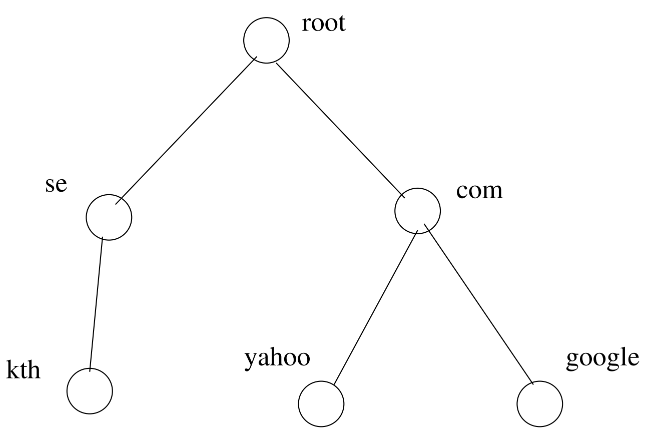


Figure 1: DNS tree

The protocol in our implementation is of course not identical to the global DNS. Global DNS uses a binary protocol over UDP, in this implementation we use an Erlang-message protocol including messages such as {`request, Pid, Domain`}, {`register, Name, Entry`} etc.

The entities in a DNS distributed system can be divided into client (makes the requests), resolver(resolves requests, queries the servers and caches responses), servers (responsible for some domain). The resolver is typically a program running on the host of the client.

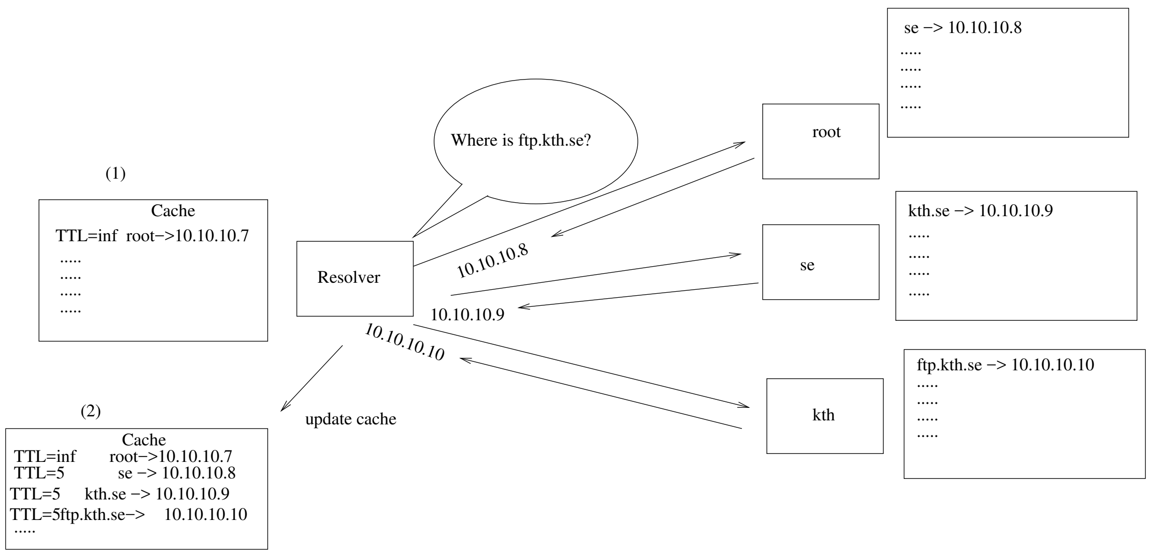


Figure 2: DNS lookup

2.2 Experimenting with the cache

In the vanilla set-up the time-to-live is zero seconds. What happens if we extend this? How much is traffic reduced?

How much traffic is reduced depends on the rate of client requests and how many sub-domains are involved. The message complexity of a regular DNS lookup through the resolver with no hit in the cache is linear proportional to the depth of the domain: $\mathcal{O}(2+2\mathcal{N})$ where $\mathcal{N}$ is the depth of the domain. The message complexity of a regular lookup at the resolver when there is a cache hit is constant: $\mathcal{O}(2)$.

What happens when TTL of cache entries is 1 min and hosts are moved while they are cached?

While the cache entry is valid the resolver will respond directly based on the cache entry which means that if a host is moved while cached at the resolver the resolver will give stale responses to lookup requests for the time the cache entry is valid.

2.3 Going further

Our cache also suffers from old entries that are never removed. Invalid entries are removed and updated but if we never search for the entry we will not remove it. How can the cache be better organised?

Some simple enhancements to the resolver cache procedure is to have the resolver go through and update the whole cache periodically at some fixed time interval to avoid stacking up stale entries forever.

How would we do to reduce search time? Could we use a hash table or a tree?

Currently the cache is on the format of `[{Domain, Expire, dns|host, Pid}]`, i.e a list of entries. Lists in Erlang are linked-lists and have $\mathcal{O}(\mathcal{N})$ access performance. This can be improved by using some data-structure offering better complexity for access (e.g constant or logarithmic) such as a map, a candidate for the key of each entry is then the hash of the domain.

Other possible data-structures could be to use a bloom filter which give constant lookup time, balanced binary trees (B-trees or AVL-trees) which give $\mathcal{O}(\log \mathcal{N})$ lookup complexity or a sorted resizable array which can be searched with binary search and give $\mathcal{O}(\log \mathcal{N})$ lookup complexity. Bloom filter should only be considered if space is a concern since it is the main perk of bloom filters. Otherwise bloom filter brings unnecessary disadvantages in the form of false positives.

3 Conclusions

This assignment gave a good rundown of how the DNS works and what kind of design choices there is to consider when deciding on the architecture of a distributed name server. In this task we pretty much followed the design of the global DNS which undoubtedly have proven to be a quite successful architecture, but one could also consider other architectures and abandon the hierarchical structure and the recursive resolving of names to a more flat architecture.

Primy: finding a large prime

Kim Hammar

June 23, 2017

1 Introduction

This reports presents the implementation of a distributed application for finding primes and highlights issues and trade-offs related to the application design and inherent challenges present in a distributed setting.

2 Main problems and solutions

- *Implementation*
  The implementation follows a basic Erlang pattern with different processes living inside message-receive loops with certain state.
- *Speeding it up*
  In this lab it has not been attempted to optimise the algorithm for checking if a number is prime or not. Rather the interesting part is how the the distribution can help improve the performance.
- *Making it robust*
  In an open distributed system there is a necessity to be able to handle byzantine behaviour and malicious processes, the main approach taken is to utilise quorum to be able to allow up to $\frac{N}{2} - 1$ byzantine worker-processes in a system of N workers.

2.1 Implementation

The processes are divided into server and workers. The server is the one orchestrating the checking for primes and keeps track of the worker nodes. The worker nodes checks if a given number is a prime on behalf of the server and reports back its findings.

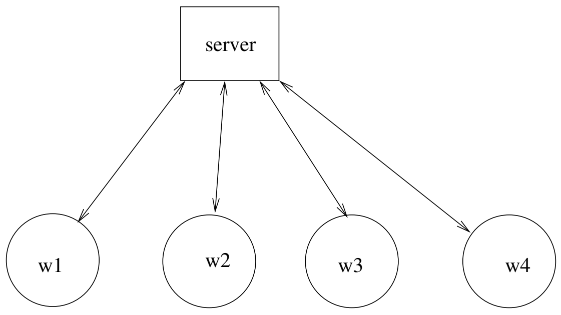


Figure 1: Server-Worker architecture

2.2 Speeding it up

An optimization for managing requests for new primes faster is that the server will keep track of which workers are busy or not and if it receives a request when all workers are busy it will itself check for primality. It is important to note that there are no fee lunch and the undertakings to make the server more robust actually have a negative affect on performance but this optimisation can help somewhat. The main bottleneck is the server waiting for responses from the workers, as we'll get back into in the following section, the server will wait for a quorum of responses from workers before deciding whether a given number is prime or not.

2.3 Making it robust

To handle dying workers we use the server as a superviser for the worker nodes. The server uses failure detectors to detect whenever a worker node has crashed (eventual failure detector $\Diamond\mathcal{P}$) and will then spawn a new worker node.

To deal with byzantine behaviour the main undertaking, as mentioned, is to use a voting-procedure where a quorum of workers need to regard a particular number as prime for the server to trust the decision and vice-verse for non-primes. This means that as long as there are $\frac{N}{2}$ correct nodes the system can deal with byzantine behaviour.

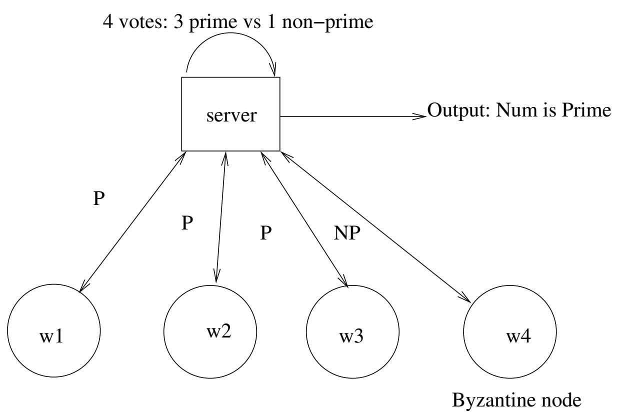


Figure 2: A quorum system for managing byzantine nodes

3 Conclusions

There is no free lunch, trade-offs is a very big part of designing distributed systems and it is important to be clear what your requirements and design goals are to know which trade-offs are worth it.

Seminar 2
Routy: a small routing protocol

Kim Hammar

September 27, 2016

1 Introduction

This report covers the work done in an assignment on distributed systems. The given task was to implement a link-state routing protocol in Erlang with the purpose of learning the construction and theory of link-state routing protocols, consistent views and problems related to network failures.

2 Main problems and solutions

The main challenges encountered for this assignment were:

- *Implementation of a link-state routing protocol and connecting routers together to a distributed system*

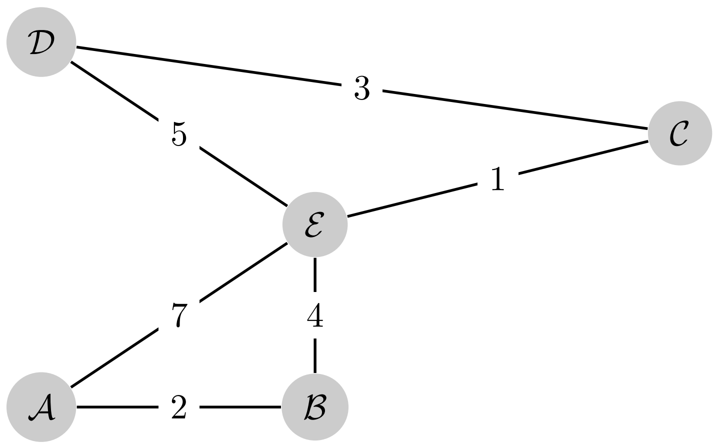
  The implementation was done in Erlang with built-in primitives for connecting nodes and message-passing between nodes.

- *Examining how the system handles network failures and how to keep a consistent view over the network.*

  Different test scenarios were examined and analyzed.

2.1 Implementation

In link-state routing every node constructs a graph-like map over the network connectivity. The map shows which nodes are connected to each other. This map is refered to as a *link-state database*. For the construction of the map to be possible it is required that routers in the network shares information about their connectivity. Routers share their connectivity information with other routers through *link-state messages*. The link-state messages are spread over the network through flooding, this is very useful since the router from where the message originated can reach a large number of nodes in the network with the update just by sending the message to its neighbours. On the other hand flooding also itroduces the risk that link-state messages are sent many times to the same node due to cyclic paths in the network. To avoid flooding of old messages, *sequence numbers* are used, where routers keep track of viewed messages and can discard (disrupt the flooding) old messages. Figure 1 shows an example of a network of routers and the corresponding link-state database (map) and routing table of one of the routers.



link-state database at router $\mathcal{E}$

$\mathcal{A}$	$\mathcal{B}$	$\mathcal{C}$	$\mathcal{D}$	$\mathcal{E}$
$\mathcal{B}/2$	$\mathcal{A}/2$	$\mathcal{D}/3$	$\mathcal{C}/3$	$\mathcal{A}/7$
$\mathcal{E}/7$	$\mathcal{E}/4$	$\mathcal{E}/1$	$\mathcal{E}/5$	$\mathcal{B}/4$
				$\mathcal{C}/1$
				$\mathcal{D}/5$

Routing table at router $\mathcal{E}$

Destination	Next hop
$\mathcal{E}$	–
$\mathcal{A}$	$\mathcal{B}$
$\mathcal{B}$	$\mathcal{B}$
$\mathcal{C}$	$\mathcal{C}$
$\mathcal{D}$	$\mathcal{C}$

Figure 1: Link-State Routing

Routers in the network are implemented as erlang processes that simulates network communication through message passing. The routers mimic an implementation of a constrained neighbour discovery protocol by using *erlang monitors*. With erlang monitors, the routers can detect failures of their neighbours. The protocol is constrained in the sense that each neighbour needs to be added manually in the first place.

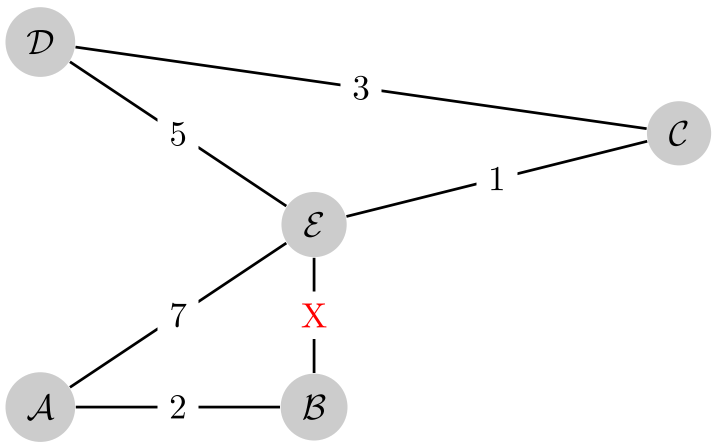
2.2 Failures and Consistency

Failures are discovered through erlang monitors, however the erlang monitor is designed particularly for process failures rather then network failures. The monitor will discover process crashes very quickly since the monitored process will send a `DOWN` message to the observer. A network failure can also be discovered since the erlang monitor will eventually throw a timeout-exception if the monitored process is not responding, but network-failures will not be detected as quickly as a process failure. In a non-erlang world, the routers would instead use some kind of heart-beat protocol to keep track of their neighbours.

Routing protocols are dependent on that routers share information with each other in order to keep up-to-date view of the network. Link-state protocols such as OSPF is based on routers sending information about their connectivity, this information is then used at the routers to keep their view of the network **consistent**. The view is then used at each router to calculate the optimal routing table. Since the routers are dependent on messages from other routers to keep their view consistent, the views are inherently *eventually consistent*, and there will be times where routers contain inconsistent views. In particular in the implementation of this assignment, views might not even be eventually consistent since the routers won't broadcast their network updates or update their routing tables if we don't explicitly command them to do so with: `router ! broadcast` and `router ! update`. When the view of the network is inconsistent at a router, the router might route messages such that they will never reach their destination or such that they take unneccesary long paths to their destination.

To improve consistency and reduce the time of in-consistency, routers should always share the information about their connectivity when a network change occurs, and even if no change have happened the information should be sent periodically to inform eventual new-comers on the network.

For example, if the link between router $\mathcal{E}$ and $\mathcal{B}$ in the network depicted in figure 1 fails, then the network should discover this failure and the routing tables should be recomputed. Figure 2 shows how the routing table and link-state database of router $\mathcal{E}$ is changed.



link-state database at router $\mathcal{E}$

$\mathcal{A}$	$\mathcal{B}$	$\mathcal{C}$	$\mathcal{D}$	$\mathcal{E}$
$\mathcal{B}/2$	$\mathcal{A}/2$	$\mathcal{D}/3$	$\mathcal{C}/3$	$\mathcal{A}/7$
$\mathcal{E}/7$		$\mathcal{E}/1$	$\mathcal{E}/5$	
				$\mathcal{C}/1$
				$\mathcal{D}/5$

Routing table at router $\mathcal{E}$

Destination	Next hop
$\mathcal{E}$	–
$\mathcal{A}$	$\mathcal{A}$
$\mathcal{B}$	$\mathcal{A}$
$\mathcal{C}$	$\mathcal{C}$
$\mathcal{D}$	$\mathcal{C}$

Figure 2: Link-State Routing

3 The world

Below you can see a map over the network used for testing. The links are by default one-way in the implementation, however in the map below a link represents a bidirectional link (I set it up that way, with two links between each connected city). The world is distributed on 8 erlang nodes and 3 different host-machines.

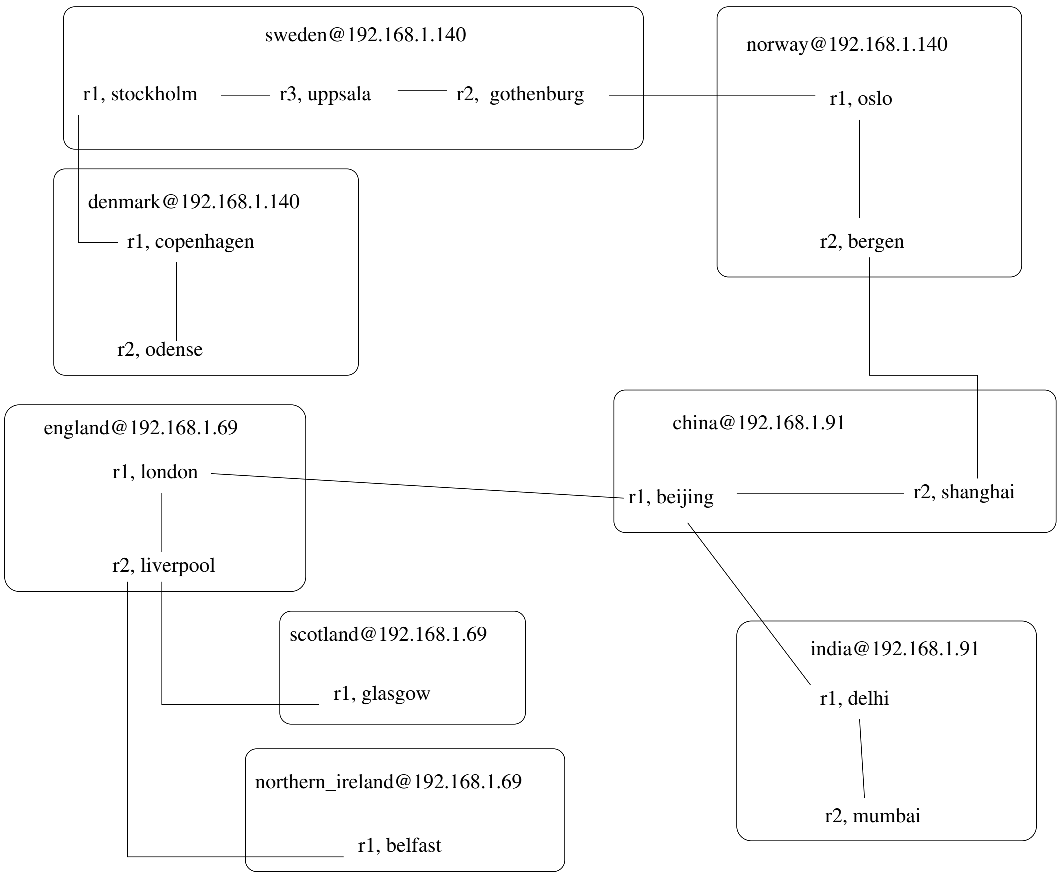


Figure 3: The world

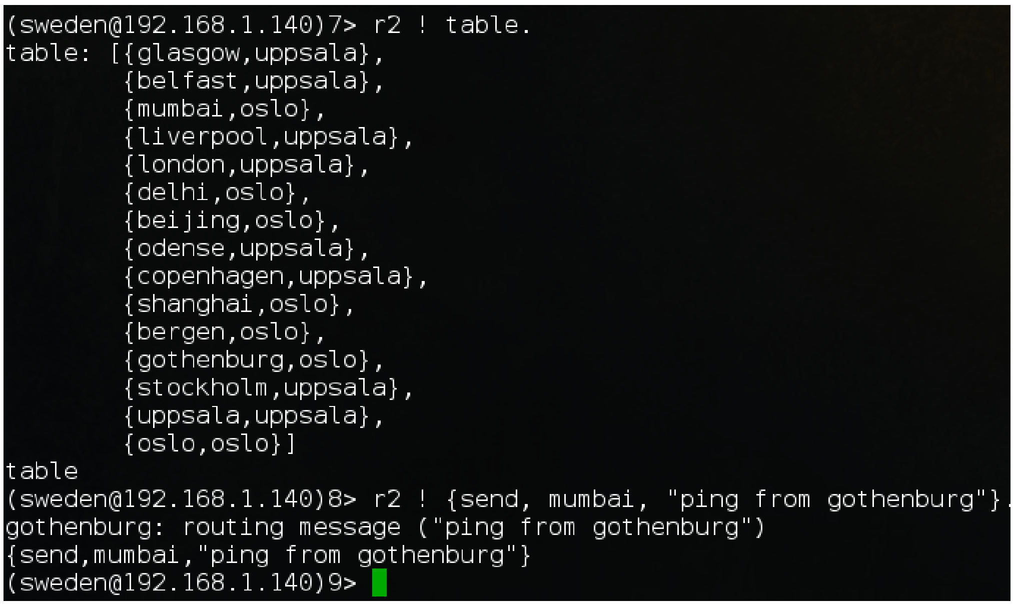
3.1 Test

A test to see if the routing in the world works and how the network handles failures.

Test scenario:
Route a message from Gothenburg to Mumbai, then kill one of routers that the used route depends on. After updating the routing tables, try to re-send

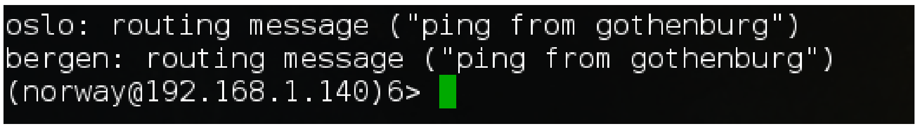
the message.

3.2 Result



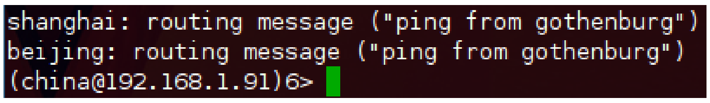
```
(sweden@192.168.1.140)7> r2 ! table.
table: [{glasgow,uppsala},
        {belfast,uppsala},
        {mumbai,oslo},
        {liverpool,uppsala},
        {london,uppsala},
        {delhi,oslo},
        {beijing,oslo},
        {odense,uppsala},
        {copenhagen,uppsala},
        {shanghai,oslo},
        {bergen,oslo},
        {gothenburg,oslo},
        {stockholm,uppsala},
        {uppsala,uppsala},
        {oslo,oslo}]
table
(sweden@192.168.1.140)8> r2 ! {send, mumbai, "ping from gothenburg"}
gothenburg: routing message ("ping from gothenburg")
{send,mumbai,"ping from gothenburg"}
(sweden@192.168.1.140)9>
```

Figure 4: Routing table at gothenburg, sweden. Gothenburg sends a message with destination address: Mumbai.



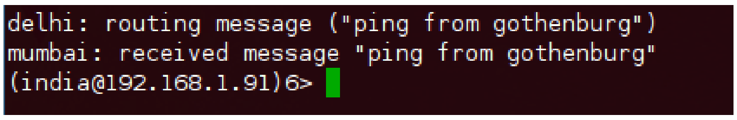
```
oslo: routing message ("ping from gothenburg")
bergen: routing message ("ping from gothenburg")
(norway@192.168.1.140)6>
```

Figure 5: The message is routed through norway. First the message arrives at Oslo, which then forwards the message to Bergen.



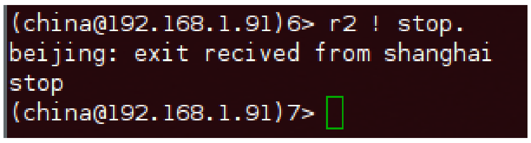
```
shanghai: routing message ("ping from gothenburg")
beijing: routing message ("ping from gothenburg")
(china@192.168.1.91)6>
```

Figure 6: Bergen routed the message to China, Shanghai. Shanghai forwards the message to Beijing.



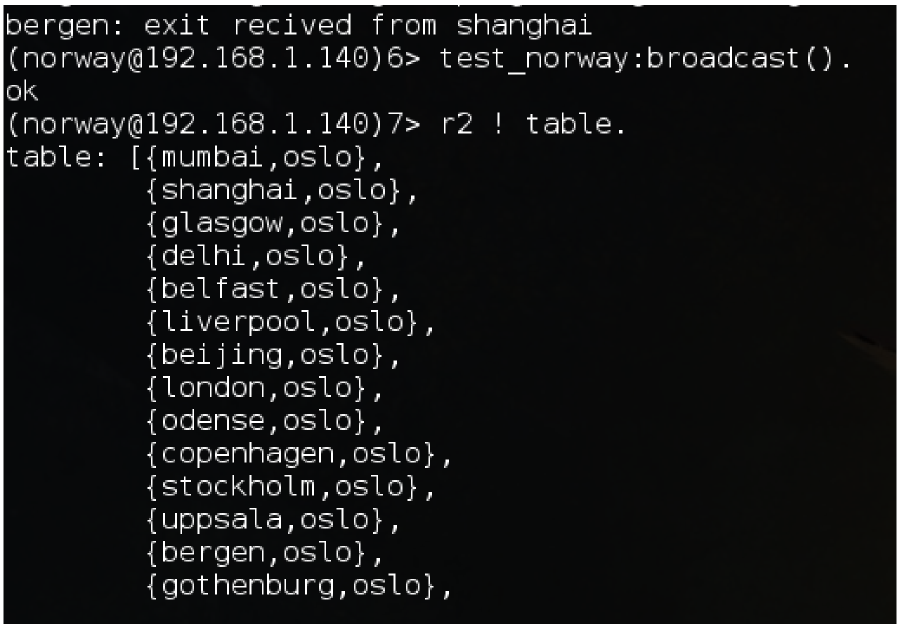
```
delhi: routing message ("ping from gothenburg")
mumbai: received message "ping from gothenburg"
(india@192.168.1.91)6> 
```

Figure 7: Beijing routed the message to Delhi, India. Delhi then forwards the message to Mumbai. Mumbai is the final destination and thus receives the message.



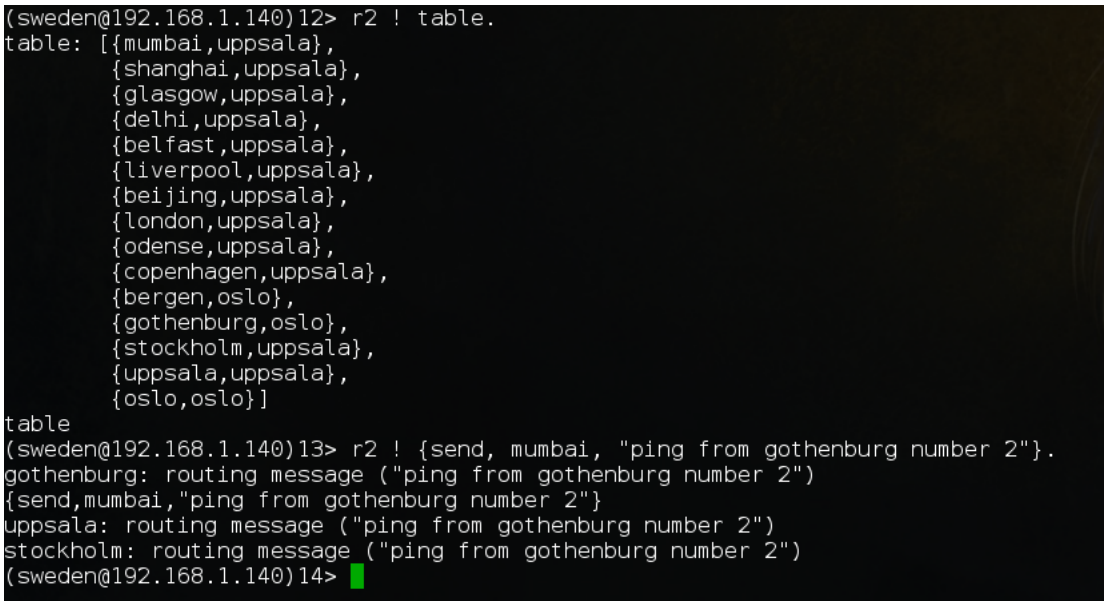
```
(china@192.168.1.91)6> r2 ! stop.
beijing: exit recived from shanghai
stop
(china@192.168.1.91)7> 
```

Figure 8: Killing the Shanghai router.



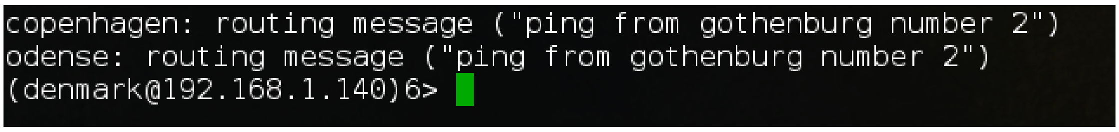
```
bergen: exit recived from shanghai
(norway@192.168.1.140)6> test_norway:broadcast().
ok
(norway@192.168.1.140)7> r2 ! table.
table: [{mumbai,oslo},
        {shanghai,oslo},
        {glasgow,oslo},
        {delhi,oslo},
        {belfast,oslo},
        {liverpool,oslo},
        {beijing,oslo},
        {london,oslo},
        {odense,oslo},
        {copenhagen,oslo},
        {stockholm,oslo},
        {uppsala,oslo},
        {bergen,oslo},
        {gothenburg,oslo},
```

Figure 9: Bergen receives a notification that the contact to Shanghai is lost. Bergen then updates the routing table and floods the information about the updated links on the network.



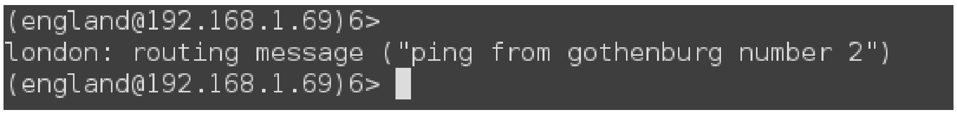
```
(sweden@192.168.1.140)12> r2 ! table.
table: [{mumbai,uppsala},
        {shanghai,uppsala},
        {glasgow,uppsala},
        {delhi,uppsala},
        {belfast,uppsala},
        {liverpool,uppsala},
        {beijing,uppsala},
        {london,uppsala},
        {odense,uppsala},
        {copenhagen,uppsala},
        {bergen,oslo},
        {gothenburg,oslo},
        {stockholm,uppsala},
        {uppsala,uppsala},
        {oslo,oslo}]
table
(sweden@192.168.1.140)13> r2 ! {send, mumbai, "ping from gothenburg number 2"}.
gothenburg: routing message ("ping from gothenburg number 2")
{send,mumbai,"ping from gothenburg number 2"}
uppsala: routing message ("ping from gothenburg number 2")
stockholm: routing message ("ping from gothenburg number 2")
(sweden@192.168.1.140)14> 
```

Figure 10: Gothenburg have received the information about changes in the network and have recomputed its routing table. Gothenburg sends a message with destination Mumbai again. However, this time the message will take a different route due to the failure of the Shanghai-router. The message is first sent to Uppsala who then forwards it to Stockholm.



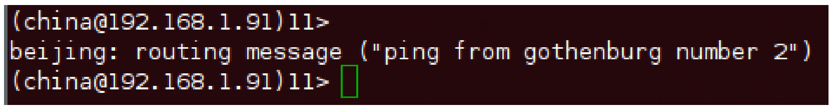
```
copenhagen: routing message ("ping from gothenburg number 2")
odense: routing message ("ping from gothenburg number 2")
(denmark@192.168.1.140)6> 
```

Figure 11: Stockholm forwarded the package to Copenhagen, Denmark. Copenhagen then directs the message to Odense.



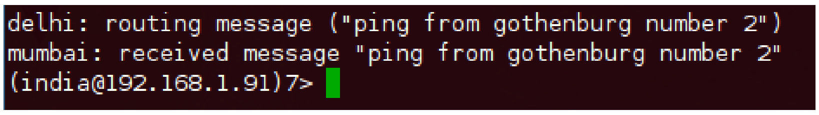
```
(england@192.168.1.69)6>
london: routing message ("ping from gothenburg number 2")
(england@192.168.1.69)6> 
```

Figure 12: Odense has a direct link to london and forwards the message there. London then routes the message to Beijing, China.



```
(china@192.168.1.91)11>
beijing: routing message ("ping from gothenburg number 2")
(china@192.168.1.91)11>
```

Figure 13: Beijing receives the message and forwards it to Delhi.



```
delhi: routing message ("ping from gothenburg number 2")
mumbai: received message "ping from gothenburg number 2"
(india@192.168.1.91)7>
```

Figure 14: Delhi recieves the message and routes it to Mumbai. Mumbai is the final destination and receives the message.

3.3 Conclusions

One of the desired properties of link-state routing is to achieve an autonomous network of routers that relatively quickly can discover failures or other changes in the network and recompute their routing tables when neccessary, before a failed link causes package loss.

When stopping a router or killing a node, the monitor discovered the failure instantly (minimal delay since on the same LAN). However when silently disconnecting one host from the network, it took ≈ 30 seconds for the other node to discover the failure due to a timeout in waiting for a response.

For this test we did not allow the routers to update their tables nor broadcasts link-state messages when they felt like it, instead this was implemented manually through a message-API, which gave us more control over the test scenario. But ofcourse in a real-world scenario this would be done automatically, where routers could send link-state message on defined time-intervals and update their tables as soon as they receive new information.

The test shows that OSPF is a quite elegant solution to a very important problem. The protocol is quite simple but gives very desirable properties of quick propagation of network changes and maintainence of routing tables. In the test we saw that the router in Gothenburg initially routed the package to Mumbai in 6 hops by going through Norway and Bergen. After the failure of the router in Shanghai this route was no longer possible. The neighbours of Shanghai noticed the failure and broadcasted the information to their neighbours, and eventually the information got propagated over the network to Gothenburg. Gothenburg could then with the updated view of the network figure out that the message can still be routed to Mumbai, but his time in 8 hops, by going the way through Denmark and England.

Seminar 1
Rudy: a small web server

Kim Hammar

September 15, 2016

1 Introduction

This report covers the work done in an assignment on distributed systems. The given task was to implement a small web server in Erlang with the purpose of getting acquainted with the procedures of working with socket API's, the client-server model and the HTTP protocol.

2 Main problems and solutions

The simple web server of interest in this assignment consists of three main parts:

- *HTTP parser*
  The server needs to parse incoming HTTP requests in order to know how to respond
- *HTTP API*
  - What to respond to certain requests?
  - What resources are available for the client? (Which URI's are mapped to the resources?)
- *Server architecture*
  - How is concurrency handled?
  - Which processes are involved? and how are the different processes related?
  - How to optimize the server's utilization of resources?
  - How to make the server scalable?

2.1 HTTP Parser

HTTP is a text-based protocol so to parse a request the server go through it character by character and compare it against the defined HTTP protocol rules. The server might receive HTTP requests in chunked parts, which introduces the problem of knowing when the full request is received. The rules of the HTTP protocol entails that the header part of the request (which comes after the request-line) is ended by a double `CRLF`. When the server have received a double `CRLF` it can parse the received headers and extract the `Content-Length` header to determine how large the body part of the request is. If the `Content-Length` header is absent, the server needs to listen until the client closes the connection since then the server cannot tell if it have received the full body of the request or not.

2.2 HTTP API

After the request is parsed the webserver decides how to respond. HTTP is the protocol for communication and since it's a request-reply protocol the server is concerned with how to *reply* to *requests*.

This minimalistic web-server serves static content that are placed in a folder of choice, if the file is not found a 404 status code is returned. The server only handles HTTP requests with the HTTP-method `GET`.

```
%%Response for a get-request to a specific URI
route({{get,[$/|FileName], _}, _, _})->
    timer:sleep(40), %% Simulate network latency
    case file:read_file("priv/" ++ FileName) of
        {error, Err} ->
            io:format("Error reading file ~n ~p ~n", [Err]),
            Body = "/" ++ FileName ++ " not found",
            http:error(Body, length(Body));
        {ok, Bin} ->
            http:ok(binary:bin_to_list(Bin), length(binary:bin_to_list(Bin)))
    end;
```

2.3 Server Architecture

When developing the server, different architectural approaches were looked at (a thread refers to a erlang process).

- *Single-threaded server* - One thread that listens for incomming requests and handles them.

- *Multi-threaded server*
  - *Worker-pool architecture* - The server creates a fixed pool of worker threads. Worker threads are programmed to listen for and handle requests when they're started. The worker-pool aproach minimizes overhead of creating new processes but have a disadvatage of inflexibility.
  - *Thread-per-request architecture* - One thread listening for incoming connections and for every connection a new thread is spawned to handle 1 request from the connection and then terminate. If the client wants to send multiple requests it is forced to open a new connection for each request.
  - *Thread-per-connection architecture* - One thread listening for incoming connections and for every connection a new thread is spawned to handle the connection. Can utilize http persistent connections to minimize TCP overhead.

Measurements where made to examine the behaviour of different server architectures, the results are presented in the sequent section.

3 Evaluation

3.1 Delimitations

Both the server and the client computer used for this test have 8 CPU cores available. The server simulates a 40 milliseconds delay with `timer:sleep(40)` for each request. The clients sends requests to the url: "/" from which the server replies with a 53-byte reply of: "`"Welcome to my simple webserver in Erlang! /Kim Hammar"`"

The point of interest for the benchmarks were how the different server-architectures behaved when dealing with concurrent users compared to a single user. Two tests with five different test cases each were examined for all the server-architectures, the cases differ on how many concurrent threads on the client-machine is sending requests to the server.

3.2 Results

Test 1: One request per connection

For this test, the client sends one request per connection. For each request the client waits for a response before it closes the connection and opens up a new one.

Client-threads	**Connections**	**Total requests**	**Server architecture**	**Time(s)**
1	1024	1024	single-thread	109.13
2	1024	1024	single-thread	103.43
4	1024	1024	single-thread	103.43
8	1024	1024	single-thread	127.14
16	1024	1024	single-thread	127.65
1	1024	1024	worker-pool (8)	108.82
2	1024	1024	worker-pool (8)	54.20
4	1024	1024	worker-pool (8)	27.21
8	1024	1024	worker-pool (8)	13.58
16	1024	1024	worker-pool (8)	10.41
1	1024	1024	thread-per-request	108.67
2	1024	1024	thread-per-request	54.85
4	1024	1024	thread-per-request	27.16
8	1024	1024	thread-per-request	14.50
16	1024	1024	thread-per-request	6.90
1	1024	1024	thread-per-connection	108.67
2	1024	1024	thread-per-connection	54.97
4	1024	1024	thread-per-connection	27.09
8	1024	1024	thread-per-connection	13.58
16	1024	1024	thread-per-connection	6.85

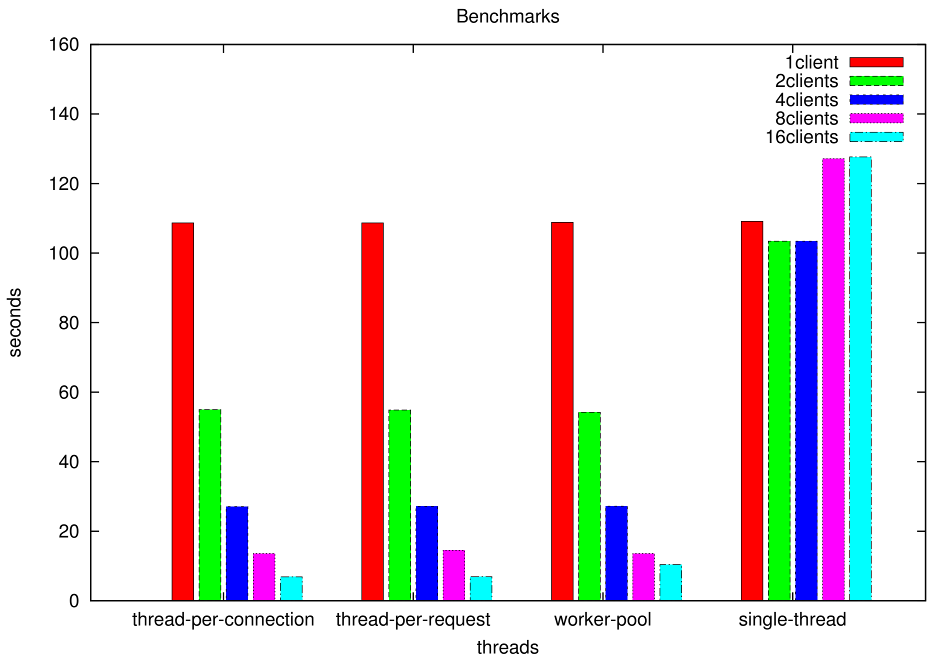


Figure 1: Benchmarks for Test1

As was predicted, the multi threaded servers exhibit a distinct performance boost over the single threaded server when dealing with concurrent users.

Further more it is interesting to see that the three different variants of multi-threaded servers perform very similar. This was also expected, considering that the clients sent only one request per connection the *thread-per-connection* and *thread-per-request* does basicly the same thing.

Also, the worker-pool architecture which consists of 8 threads (same as the number of cores on the server machine) reaches basicly the same throughput when dealing with 8 concurrent clients as the servers that creates threads dynamically. When there are 16 concurrent clients the results show that the servers that create threads dynamically give slightly higher throughput than the worker-pool.

Test 2: Client tries to send all requests over the same connection A final benchmark was made to examine how the overhead of creating new TCP connections affect the performance. In this test the client will try to send requests over the same connection rather than closing the connection after each request (this is often the case in real-world scenarios of HTTP communication, where browsers use HTTP-persistent connections).

Bare in mind that a prerequisite for sending multiple requests over the same connection is that the server keeps the connection open. If the server closes the connection, the client is forced to open up a new one in order to communicate.

Client-threads	**Connections**	**Total requests**	**Server architecture**	**Time(s)**
1	256	256	thread-per-request	27.301071
1	512	512	thread-per-request	55.17
1	1024	1024	thread-per-request	109.120926
1	2048	2048	thread-per-request	217.15903
1	1	256	thread-per-connection	26.85301
1	1	512	thread-per-connection	52.07
1	1	1024	thread-per-connection	105.558201
1	1	2048	thread-per-connection	203.094831

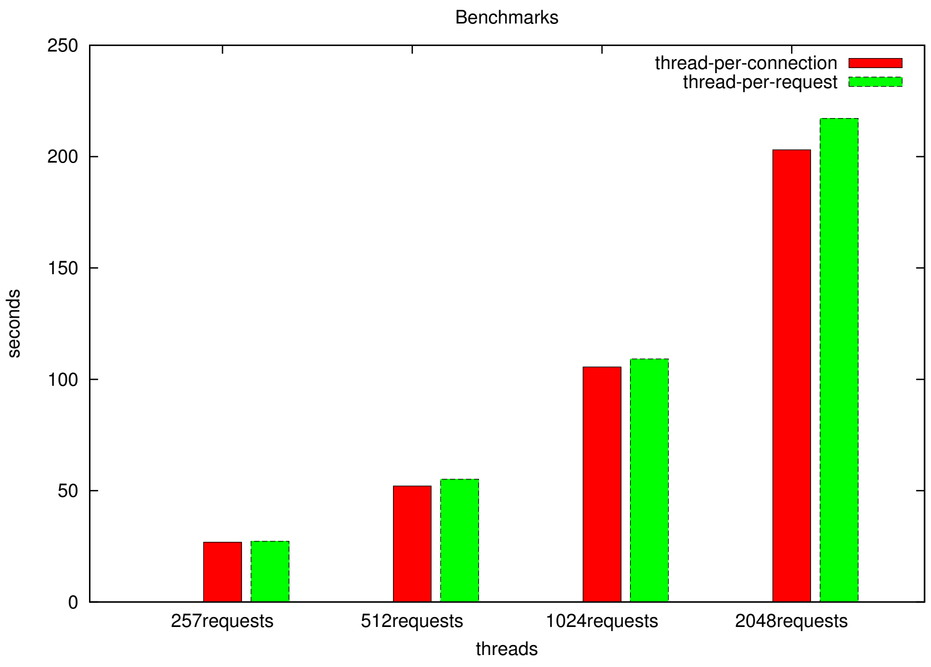


Figure 2: Benchmarks for Test2

The results demonstrate that the performance difference of opening a new connection for each request compared to reusing an old request is very modest. There is however a slight difference which increases with the number of requests.

4 Conclusions

The results manifest that to fully utilize a multi-core architectured hardware you can advantegously use a multi-threaded architecture over a single-threaded architecture.

As a result of the delimitations mentioned, we cannot draw precise conclusions of wether a worker-pool is beneficial over dynamic creation of threads. The results from the limited benchmarks show that dynamic creation of threads was benifical in this case. This is likely because even if the upper-bound on parallel tasks on the server machine is 8, having more erlang processes than 8 can hide some latency and improve performance that way. The latency that can be hidden is that of context-switching between handling a connection and listen for a new connection, however the dynamic creation of threads architecture is more vulnerable than the worker-pool architecture to a potential denial-of-service (DoS) attack. The dynamic server will happily try to create billions of threads, which will eventually cause it to crash.

The results also testify that, when possible, TCP-connections should be reused in order to reduce the overhead of creating new connections. Even though the overhead experienced in the tests was supringsingly small in proportion to the number of requests.

Principially the results can be regarded as rather credible (considering the stated delimitations), the tests were ran multiple times and showed off basicly the same results every time except for one parameter; The single-threaded server differed on ± 10 seconds when dealing with 1024 requests from $8 - 16$ clients. It's hard to say why. The rest of the results only differed at most ± 1 second.

For all of the tests made, the client got 0 timeouts from the server.

Snapy: the search for dead marbles

Kim Hammar

July 8, 2017

1 Introduction

This report presents the work done in an assignment on distributed systems. The assignment entailed to implemented a snap-shot algorithm in Erlang. The algorithm resembles the Chandy-Lamport algorithm and constructs a global snapshot of the system state, this state is then used to perform garbage collection of marbles.

2 Main problems and solutions

- *Implementation*
  Implementation was a bit tricky for this assignment since I found some inconsistencies in the code-snippets provided and the purpose of the GUI was poorly described. As a result the GUI was designed from scratch and the result is depicted in the figure below.

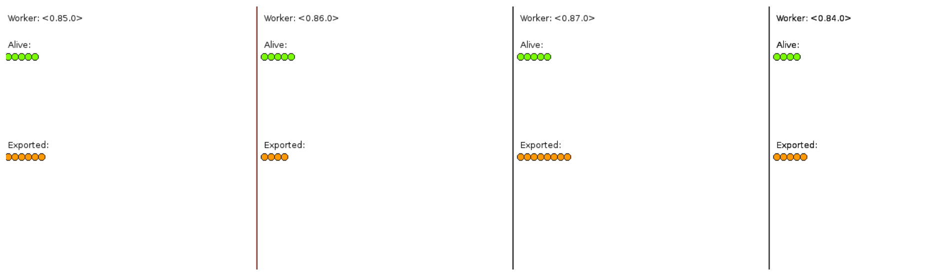


Figure 1: GUI

- *Tests* The GUI was very helpful in conducting verdicts when testing the system and measuring the effectiveness of the garbage collector.

3 System Architecture

Every worker in the system knows about each other so the processes and communication channels take the form of a fully-connected graph. In the figure below an instance of the network with four workers is shown.

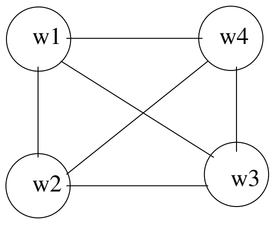


Figure 2: Network of processes

In the snapshot-algorithm used (variant of Chandy-Lamport) the state of messages in transit is included in snapshots by the use of “markers” which works as a flush of the communication channels.

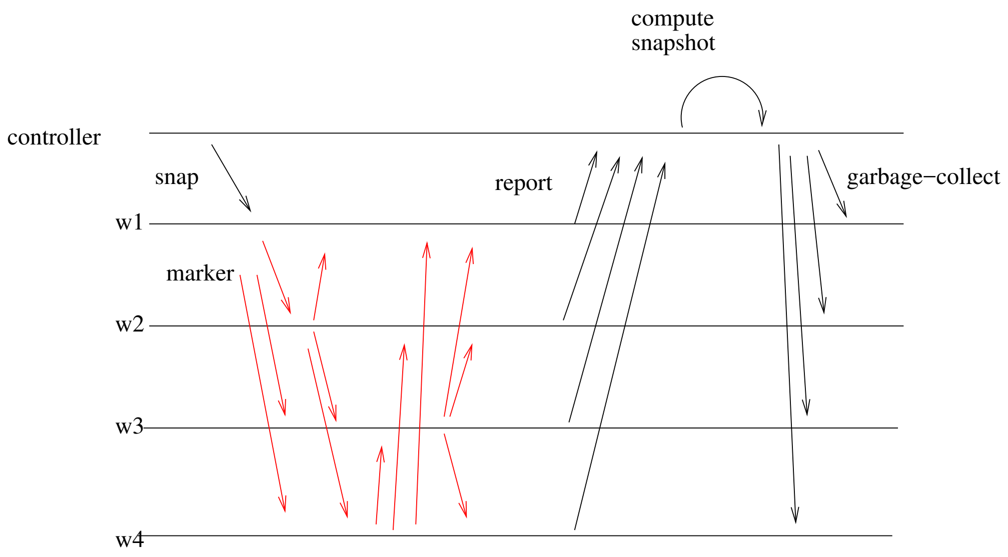


Figure 3: Snapshot algorithm

4 Tests

After testing different versions of the snapshot-algorithm it is clear that the final version which too takes in account the messages in transit when taking the snapshot is most effective. The first version of the snapshot-algorithm which didn't take messages in transit into account could in some cases, depending on the network latency (simulated), garbage collect marbles prematurely.

The downside of the snapshot-algorithm that resembles Chandy-Lamport is that it has very high message-complexity which makes it more expensive to perform.

5 Conclusions

Computing snapshots of the global state in a distributed system is not as easy as just aggregating each process local state, one much take messages that are in transit on the channels into account as well. The Chandy Lamport algorithm is a classic algorithm for computing this by using so called "markers".

Garbage collection in a distributed system is complex and computing a global snapshot is not the only way nor the most efficient way of doing it. Global snapshots is useful in more settings than just garbage collection, it can be used to break deadlocks or just general debugging of distributed systems.