The Design of a Replicated Local Data Store

Linus Ländin

December 20, 2009
Master's Thesis in Computing Science, 30 ECTS-credits
Supervisor at CS-UmU: Stephen J. Hegner
Examiner: Per Lindström

UMEÅ UNIVERSITY
DEPARTMENT OF COMPUTING SCIENCE
SE-901 87 UMEÅ
SWEDEN

Abstract

Distributed systems consists of multiple computers (nodes) that communicates through a computer network. In distributed systems it is often desirable to have the nodes share information in a consistent manner. In some special cases it is desirable to have all nodes contain exactly the same information at all times.

This report describes a replicated local data store called RLDS which is a general purpose caching and replicating system that supports in-memory and on-disk caching as one logical unit. The proof-of-concept system described in this thesis is intended to be used with trading systems at a financial company.

Acceptance testing shows that the proof-of-concept system works in a consistent manner, however performance wise there is still room for improvements, for example, object serializing could be optimized.

Contents

List of Figures

Chapter 1

Introduction

This report describes a replicated local data store called RLDS. RLDS is a general purpose caching and replication system that supports caching in-memory and on disk as one logical unit, the data store. RLDS instances running on different hosts are connected via a group-communication layer and the data store is replicated amongst these hosts allowing all hosts to have an exact copy of the data store. Objects can be added to the store from any host and all hosts in the communication group will store the object in its local data store ensuring that all hosts have the same objects. Applications communicate with the RLDS through a generic application programming interface (API) and the focus of the system has been on making it generic and reliable and within that making it fast.

The RLDS is intended to be used by online marketing systems (OMSs) at Nomura to provide a common data store where they can store and retrieve serializable Java objects in a replicated and consistent manner. RLDS is written intirely in the Java programming language to ease the integration with the existing OMSs and other systems at Nomura which would be users of the system.

The work has been done at Nomura Sweden AB located in Umeå. Nomura Sweden AB is a part of the global investment bank Nomura Holdings Inc., which in turn is member of Nomura Group, one of the major industrial and financial conglomerate groupings of Japan.

Nomura has large systems that handles orders from their customers to different stock markets. Nomura has many different customers that perform tradings in different markets which respectively have different requirements on number of transactions, time taken to perform a transaction et cetera.

In Chapter two the problem is described along with the background, purpose and method to tangle the problem. Chapter three shows the design for reliability and distribution of data and takes the reader through the implementation of the design. Chapter four shows some conclusions of the thesis and Chapter five acknowledges the persons who helped me throughout this thesis. In the appendices the reader can find a user's manual, the evaluation and testing procedure and algorithm description.

Chapter 2

Problem Description

This chapter gives background to the problem the purpose of the solution, what is wanted and what it will solve.

2.1 Background

At Nomura, direct market access systems (DMAS [5]) handle large amounts of data that are sent to the systems in a high pace. The messages received are typically market orders and must be handled fast and efficiently. The overall system consists of at least two data centres (two as of today), where one data centre is working as the primary and the other works as secondary. The data centres run OMSs, Online Market Systems [12], which allows customers to execute trades on market systems such as the Nordic stock exchange OMX.

The data centres contains multiple OMSs which handles different types of trading and on different markets. Nomura has different customers that performs different trading on different markets. This means that customers that performs day trading have high speed demands on transactions and they need to be carried through immediately. On the other hand, other customers may only need the transaction to be carried out during the day.

2.1.1 Example of the Current System

An example describes the flow of the system. An order come in to the system to the OMS, the OMS processes the order and then sends the order ro external markets. An order reply is returned to the OMS and later on an execution report from the market is returned to the OMS, i.e. a confirmation that the buy has been carried out. Confirmation messages are also sent to the customer who issued the order.

The example in Figure 2.1 illustrates the typical system where order management is divided into two instances, one which handles orders beginning with the letters A to M and one which handles orders beginning with the letters N to Z. These two instances have a backup running at another data centre, typically not located in the same building. As of today, both data centres are located in London and the communication within a data centre and between data centres is 1 Gbps. The average latency between the two data centres is, as of today, 0.147ms. Minimum and maximum are 0.117ms and 23.837ms respectively. Under normal conditions the orders come to the primary data centre and the backup is only used when a crash occurs.

One OMS consists of multiple components which can be divided into two groups, one that has high demand on speed that performs the actual trading and one that runs at a lower priority

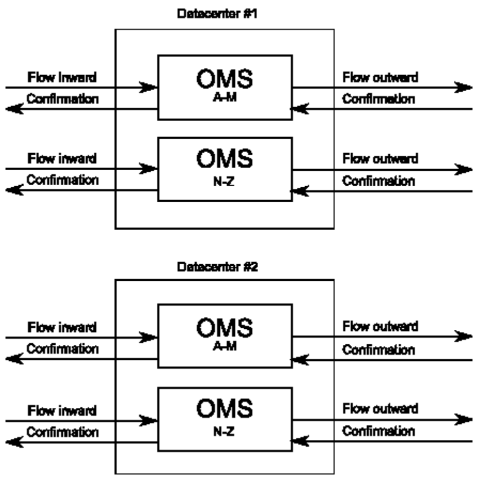


Figure 2.1: Example of the system as it is today

that runs “after” the high speed ones creating reports, logs and such. Some components are dependent on other components or depends on other components as part of a sequence through the OMS.

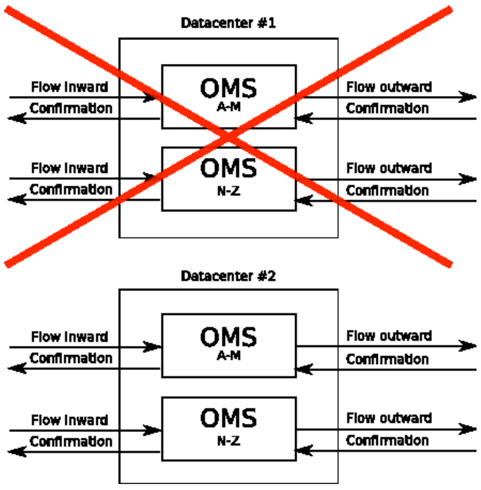


Figure 2.2: Crash of entire primary data center

2.1.2 Crashes

Since the system is quite complex there exists a number of possible scenarios for the system, or parts of the system, to crash. The first crash would be that a whole data centre goes down, as illustrated in Figure 2.2. If the whole primary data centre crashes, the secondary, backup, data centre has to manually be set “in charge” as the primary.

Another possible crash is that a computer that hosts an OMS crashes. This is illustrated in Figure 2.3 with the OMS from A to M crashes. This crash is similar to the one mentioned above with the one exception that only the crashing OMS from data centre 1 will be replaced by setting the backup OMS in data centre 2 in charge. With this scenario the OMS from A-M will be running on data centre 2 and the OMS from N-Z will be running on data centre 1.

In the example above, if the OMS from A-M crashes on data centre 1 and the OMS from N-Z crashes on data centre 2 the OMS A-M will run on data centre 2 and the OMS N-Z will run on data centre 1. If OMS A-M crashes on both data centre 1 and 2, only OMS N-Z will continue running on data centre 1.

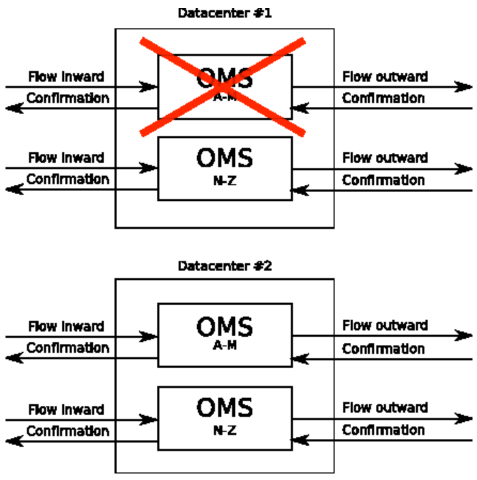


Figure 2.3: Crash of entire primary OMS

A third scenario is that one or more components in an OMS crashes. When this happens, that OMS's backup in data centre 2 will be started and now we have two OMSs located in two data centres that does 50% of the work each. An illustration of this type of crash is found in Figure 2.4. The green boxes are the components that are running.

It is possible to have multiple crashes of the same or of different sorts. These will be handled as combination of the crash recovery techniques described above. Similary, if all hosts crash at both data centres this will be the same as if the whole data centres came to a halt. However, there is no protection against the crash of both data centres.

2.2 The Need for Replication

Nomura wants their different systems to be as fast as possible while guaranteeing synchronization and validity of data within their system. It is not just important for the system to be fast while running in normal operation. If an OMS crashes its backup needs to be up and running as soon as possible. A solution to this is suggested via the use of a replicated local data store. This makes it desirable to have the data in data store replicated in both data centres. For illustration of the desired system see Figure 2.5. The grey part is the RLDS and is what is designed and constructed as a result of this thesis.

The figure illustrates the API, the OMSs, and other applications which use the RLDS, the store (in-memory and on disk) and the communication between instances of RLDS on different

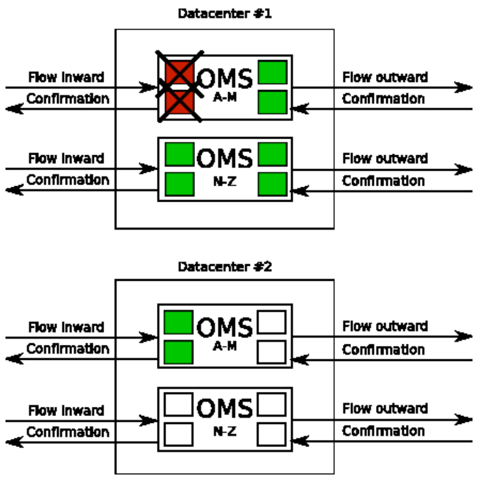


Figure 2.4: Crash of components as part of the OMS

hosts. Since the store is modelled as a single logical unit any host can run the in-memory and choose whether or not to run the disk store.

The goals of the proof-of-concept system are:

– Provide a data store that is accessible from different data centres;

– Provide a generic interface that not only OMSs but other applications and systems can use;

– Support insertion of data from different data centres and synchronization between the hosts at these data centres;

– Handle crashes of hosts and data centres as described previously.

Nomura wants a system that is generic and that can be used in more areas than the one described previous in this chapter. Thus, the proposed system will have an interface through which OMSs as well as other applications can access the system. Existing related products to this system include Ehcache [6], Terracotta [13], Velocity [10], to some extent memcached [7]. Neither of these where chosen because Nomura wants a custom made system that they can have total control and ownership of.

2.2.1 Types of Data

Two primary classes of data are to be stored in the data store: customer information and order information. Customer information is about customers and is only needed to be stored in memory, not persistently to disk because this customer information is saved in a separate database. The data are located in another database and the data store is either primed with the data or entries are fetched from their database at lookup. Customer information can, in addition to the static data, contain real time data such as the number of orders placed by that customer or total profit et cetera and are normally used by the OMS to determine whether an order is valid or not.

Order information is created in real time and is primarily contained in memory, but is replicated to the alternate data centre where it is stored persistent on disk. The persistent data on disk is used when system crashes occur.

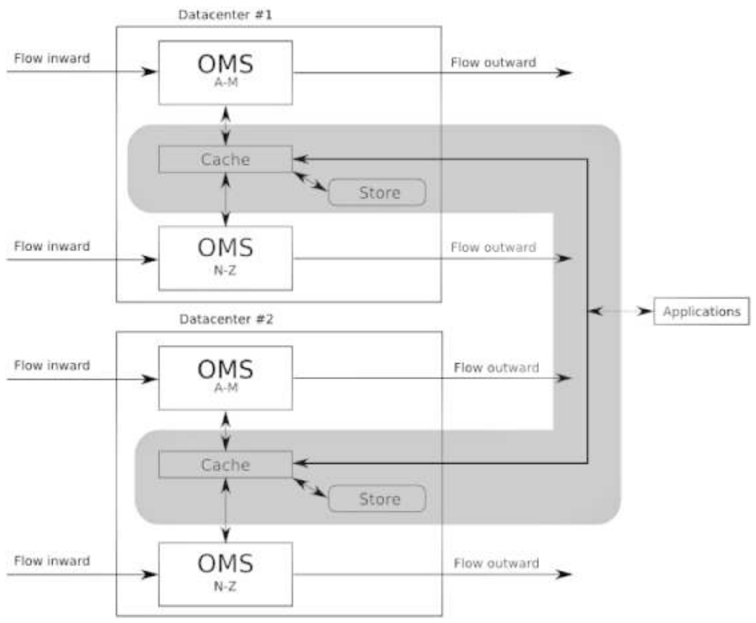


Figure 2.5: Example of the augmented system

Both these types of data need to be modifiable by the OMSs and they also might need to be manually modified and this is done by an administrative application which uses the API.

2.2.2 Handling of Crashes

To ensure that the replicated data store can handle the crashes described above, synchronization between data stores on different data centres is required. In the most common case data centre 1 will be the primary, with the data in the store replicated to data centre 2, where the data is stored on disk. If both data centres crash, the data store must restart using the data that is persistent on disk.

There is one instance of the RLDS active on each host and these are connected to each other. Each of these can be individually configured to have its copy of the data store only in memory or both in memory and on disk. This is further explained in Chapter three.

The system is configurable to either recover from disk in under a minute or recover from disk in under 30 minutes. When recovering in under a minute the data from disk is not cached in memory which it is when recovering in under 30 minutes. Apart from these time constraints, the data store operates as fast as possible.

Chapter 3

A Design for Reliability and Distribution of Data and Its Implementation

In this chapter the design and implementation of the Replicated Local Data Store (RLDS) is described. The RLDS is written in the Java programming language. The system has been designed and implemented with consideration to the methodology and technology described in the Distributed Systems course given by the department of computing science at Umeå university. The different parts of the design and the overall architecture of the system are illustrated in Figure 3.1 and described more in detail further on in this chapter.

The design consist of an application programming interface (API) to serve as the generic interface for the OMSs and other applications. It consists of two storage modules, for storing objects in primary memory and for storing objects on disk. It also has a group communication module (GCM) that supports reliable and totally ordered messaging between instances of the data store located in the same or different data centres. The most important module of the system is the controller. The controller is responsible for synchronizing data between instances. The overall implementation is illustrated as an unified modeling language (UML) diagam in Figure 3.2.

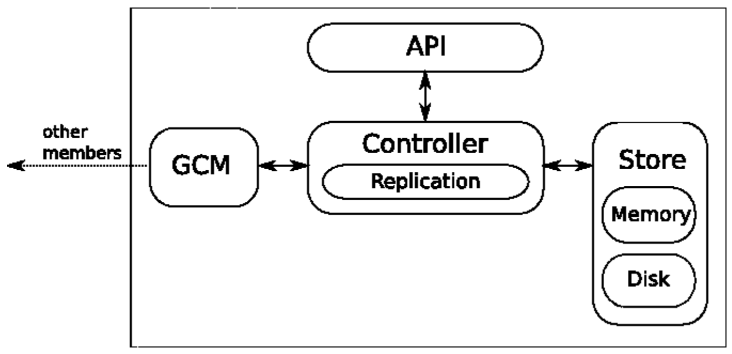


Figure 3.1: Overall architecture of the design

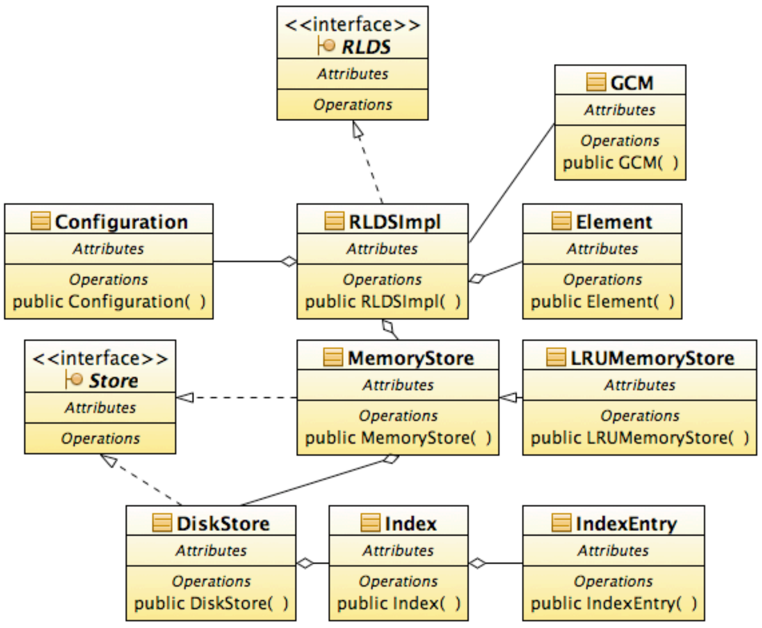


Figure 3.2: UML diagram of the overall system

Here is an example to illustrate how the system should work. Take the example described in Chapter 2 (Figure 2.5) where two OMS ran at one data centre with backups running at a second data centre. To allow these OMS to have access to the same data all OMS run an instance of the RLDS connected to each other via the group communication module (GCM). All they need is to be part of the same process group. If both OMSs on both data centres are members of the same group their respective data store will look identical.

OMS A-M in both data centres is running the RLDS and is part of group A. OMS N-Z in both data centres is running the RLDS and is also part of group A. If OMS A-M in the primary data centre adds an object O both the OMS N-Z in the primary data centre and the OMSs in the secondary data centre will add object O. The same goes for updates and deletion.

3.1 Application Programming Interface

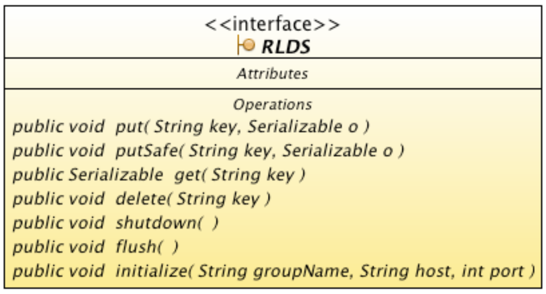
The application programming interface (API) provides a generic interface that the OMSs and any other applications can access the data store through. The API handles the communication between these components and the controller module. The interface supplies methods for insertion of objects, accessing objects and deletion of objects. In accordance with requirements from Nomura, the API contains a method (putSafe) for putting an object into the data store that will guarantee that the object is stored. This method will not return to caller until it has completed the put action.

The UML description of the interface can be seen in Figure 3.3. The interface is designed to be as simple and self-describing as possible. All that is needed to store an object is to make sure it is serializable and a unique key in form of an String along with it. A serializable object is an

```
// Assuming the rlds has been set and initialized
// RLDS rlds = ...
// Creating a key
String key = "key1";
// Creating a serializable object
Serializable object = new SomeSerializableObject();

// Store the object associated with key
rlds.put(key, object);

// Get the object associated with key
Serializable object2 = rlds.get(key);

// Delete the object associated with key
rlds.delete(key);
```

Listing 3.1: Accessing objects through the API example

object that allows for transforming the object into a stream of bytes and from a stream of bytes back into an object. This is also known as marshalling and un-marshalling. The system utilizes native support in Java for serializing and de-serializing objects.



```
<<interface>>
RLDS
Attributes
Operations
public void  put( String key, Serializable o )
public void  putSafe( String key, Serializable o )
public Serializable  get( String key )
public void  delete( String key )
public void  shutdown( )
public void  flush( )
public void  initialize( String groupName, String host, int port )
```

Figure 3.3: Application Programming Interface to the RLDS

The key associated with the object used when storing an object into the RLDS is used for accessing that object from the RLDS as well as deleting it. See Listing 3.1 for an example of how to access objects making use of the API.

There exists another method for putting objects into the RLDS, putSafe. This is a blocking method that flushes the object to the replication part and to the in-memory store as well as to disk store and will not return until the object is properly stored. The arguments to putSafe is the same as for put. See the User's Guide in appendix A.

The interface provides another functionality for ensuring that entered objects are stored properly and that is the flush method. It can be used whenever you want to make sure that the objects put into the RLDS are properly stored and replicated. This method is always called internally to ensure safe operation and is also blocking. If the RLDS is configured without disk store this method returns immediately.

The API provides two methods for starting up and shutting down the RLDS. These are initialize and shutdown respectively. The initialize method is used to connect the RLDS to the group communications manager. The shutdown function is to be called when the system is shutting down and performs flushing of objects and recursively shuts down different modules of the system such as GCM and data stores, et cetera.

3.2 Store

The store is responsible for storing the objects, in-memory and/or on disk. The in-memory store is designed to be used with different cache eviction rules such as Least Recently Used (LRU), Least Frequently Used (LFU), Most Recently Used (MRU) or First In First Out (FIFO) to handle the case when system memory runs out. Currently, in the proof-of-concept system only LRU is implemented. The disk part handles storing the objects on secondary disk and also handles retrieving objects from disk after a system crash or in the case that object is evicted from the system memory and need to be retrieved.

The store is the interface between the controller and the in-memory store and between the in-memory store and the disk store. The store interface has many similarities with the API and contains five public methods: put, get, delete and flush for storing, accessing, deleting and flushing to secondary storage, respectively. The fifth method, shutdown, is used to make cleaner exists when the system is shutting down. These are used by the controller to forward requests from the calling program or from the replication part. Put takes an object of type Element which is the element to be stored and get and delete takes a String representation of the key associated with an Element.

The class Element is the wrapper class for serializable objects put into the store. The Element objects holds a serializable object as payload, a string representation of the key, a hit counter and a timestamp for when the Element last was accessed. The hit counter and the timestamp are updated every time the payload is accessed and are used to implement different kinds of eviction policies for the memory store. As mentioned above, the memory eviction policy implemented in the proof-of-concept system is the LRU policy.

In addition to the attributes mentioned above, the Element class contains a view index number that dictates which group view the Element was stored in. A group view is set of group members at a given point in time that contains information about which members where a part of the group at that particular time. Group views are associated with an integer value.

3.2.1 Memory Store

The in-memory store is a cache for the disk store or stand-alone if no disk store is present and is simply a hash table that maps keys to wrapper elements. The in-memory store contains the same index as the disk store. However, due to disk size being larger than random access memory the in-memory store may only contain a subset of the disk store. It implements the store interface and forwards calls to the disk store if there is any present.

Apart from the store interface the in-memory store contains a method for accessing all the Elements from a specific view. This is done by the getAllFromViewIndex method.

The Least Recently Used (LRU) eviction policy is the policy implemented in the proof-of-concept system. The eviction is only intended for the in-memory store and not for the disk store since the disk space is considered to be enough on most machines. However, the eviction policy should not be needed since the amount of data is considered to fit in physical memory of intended servers.

3.2.2 Disk Store

The disk store is an extension of the in-memory store by supporting storing of Element objects persistently on disk.

Incoming requests to the disk store are queued in a worker queue that is handled by a worker thread that writes these changes to disk. The disk part is divided into two files, a *data file* and an *index file*. The data file contains the serialized elements and the index file contains the key to the element and the position and size to the serialized object in the data file. The worker thread acts as a spool by writing objects from the queue to disk as fast as possible. The worker writes the index file after a given timeout or when the flush method has been invoked. The spool is needed because it takes more time to access disk than memory and without a spool requests to the disk will block the system. The disk store needs both an data-file to store the elements and an index-file to keep track of were the actual data is stored. The index and data file names are specified in the constructor of the RLDS implementation RLDSImpl.

An illustration of the store is found in Figure 3.4.

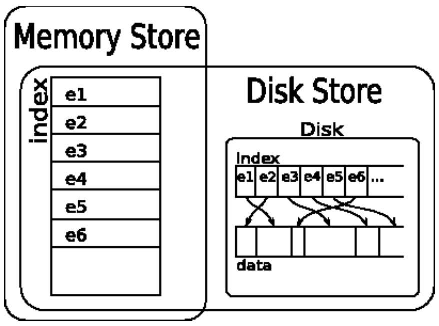


Figure 3.4: Illustration of the memory and disk store

3.3 Group Communication Module

The group communication module (GCM) is used to provide a common communication layer for the proof-of-concept system running on different hosts on different data centres. The GCM provides process groups with group management, view synchronization of group members and totally ordered reliable multicast. A process group is a set of members that upon signals to the group all members in the group will receive that signal.

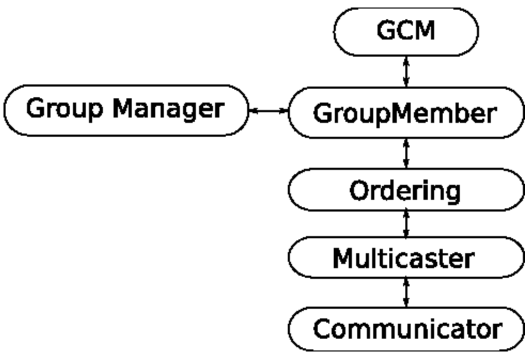


Figure 3.5: The layering structure of the GCM

The GCM consists of a group manager that handles creation, joins and leaves from groups and is divided into several layers to enforce loose coupling and modularity. These layers are group member layer, ordering layer, multicaster layer and communicator layer, as illustrated in Figure 3.5. The group member is a member in one group and a GCM instance can have many group members since it can be a member of several groups. The ordering layer is responsible for ordering the outgoing and incoming messages. The total ordering is achieved by one group member is elected through the Bully Algorithm [1] to be the sequencer of the group. Every message is sent to the sequencer which decides the order of the message and sends it to the rest of the group ensuring total order of messages throughout the group.

The multicaster layer achieves reliable multicast through keeping a list of the latest received messages and broadcasting incoming messages not found in the list once. This means that all members will send every message to all other members as described in [4]. The communication layer handles the communication between group members. The communication channel used is TCP over IP.

All messages are sent through the GCM so that if the group changes the GCM will notice this and handle the group management involved in this. This way the view of the group can be kept coherent and synchronized, as in view synchronous systems [2]. The GCM interface contains methods for creating, joining, disconnecting and removing groups. As well as sending messages to the group and registering for messages and group view changes. The interface also has a method for connecting to the group manager.

3.3.1 Group Manager

The group manager is a central point discovery service that is used to connect members to a group. The group manager is responsible for delivering consistent views of the group to the groups members. The group manager acts as a name server in the system and handles the initial connection between group members in a group. The group manager is only needed to set up a group and when group members have joined the group the group manager could crash and the group would still function. If the group manager has crashed no new members can join the group.

3.3.2 The Object Group Design Pattern

The object group design pattern is an object behavioral pattern for group communication and fault-tolerance in distributed systems [9]. Events within the object group honor the Virtual Synchrony model [3]. In virtual synchrony multiple programs in a network can be organized into process groups. Messages are sent to the process group instead to individual programs where each message is delivered to all the group members and in the same order. Owing to virtual synchrony, the size of an object group can vary at run-time, while client applications are interacting with the object.

Take the example where we have a network phone book service spread across multiple hosts, see Listing 3.2. The client applications have one object reference that represents the entire group, and don't need to worry about references to a particular group member. The install and remove operations are multicast to the whole group since they change the replicated state of the directory object. The lookup operation does not need to be replicated since it does not alter the state and can be performed on a single member.

A request to the object group will return as long as at least one group member is alive. When a new member wants to join the group operations on the group will be suspended. Then

[1]The bully algorithm is an election algorithm used in distributed systems. For details see Appendix B.

```
interface Directory {
    void install(String key, Object value) throws EntryExistsException;
    Object lookup(String key) throws NoSuchEntryException;
    Object remove(String key) throws NoSuchEntryException;
}
```

Listing 3.2: Sample network phone book service

the internal state of one group member is transfered to the new member. This could be tedious since the interal state contains the whole data store. After that group operations are resumed. This procedure is fault-tolerant because if the member which supplies its internal state fails another member will take over (and redo from start), and if the new member fails it will be discarded and the group will carry on as before.

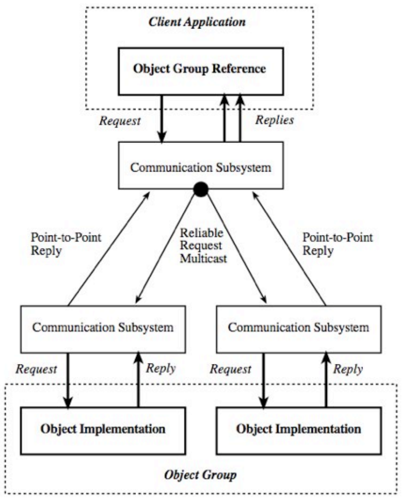


Figure 3.6: Structure of the Object Group Pattern

Structure

Figure 3.6 shows the overall structure of the Object Group Pattern. The different parts are; client application, object group reference, communication subsystem, object implementation and object group. The client application creates an instance of an object group reference and binds the reference to a target group by using a name server to perform the name-to-object mapping. After that, operations issued through the reference will be carried out on the object group.

The object group reference is a surrogate object with the same interface as the group members. When a member function is called on the group reference, the call will marshall the arguments, multicast the request to the group members, wait for a response, unmarshall the response and return the result to the calling client.

The communication subsystem provides a low-level interface to reliable multicast, group management, and light-weight processes. For the sake of flexibility and portability, the communication subsystem implements a generic interface onto the real interface provided by the actual communication system implemented.

An object implementation provides the functionality of the service, e.g., it implements a phone directory. Each group member implements the same interface. The object implementations form a logical object group. Through reliable multicast each group member are ensured to receive all requests client applications have issued through the object references. Group members are ranked based on their life-time, where the oldest receives rank 0, next oldest rank 1 and so on. When an object joins or leaves a group all members obtain a view change notification to inform about new ranks, number of members et cetera.

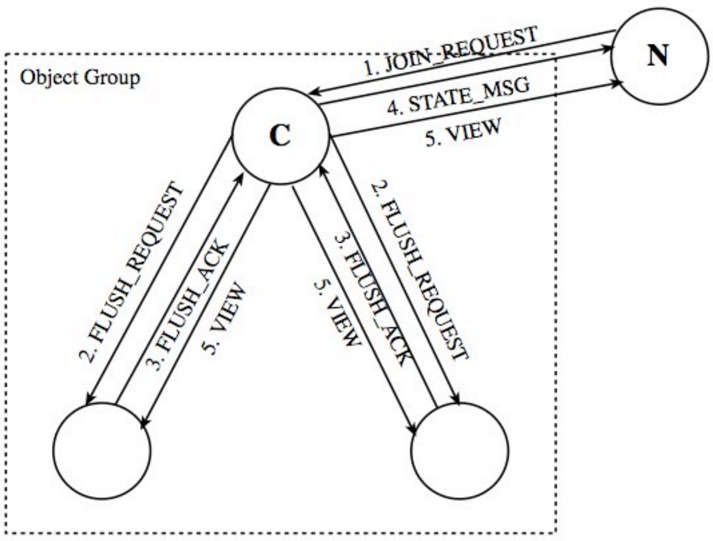


Figure 3.7: View Management Protocol of the Object Group Pattern

View Management

A member of a group always knows which members are in the group through the use of *views*. A view is an ordered set of object references, one per group member, ordered from the oldest member, the next oldest, and further on to the newest member. Each member knows its own rank and the ranking is consistent at all members. Joins and parts from a group forces the change of view at all members in the group. View management is an important component of the Virtual Synchrony model [3] and the object group pattern view management protocol is illustrated in Figure 3.7.

In Figure 3.7 object N requests to join a group containing three members where object C is the coordinator. The coordinator is always the oldest member. N sends a JOIN_REQUEST to C which in turn sends a FLUSH_REQUEST to all group members to ensure that cached requests are delivered to the client in the old view. All members respond to C with a FLUSH_ACK when done. Now user requests are held back until the new view is installed. When C has received all acknowledge messages from the group it sends a STATE_MSG to the new member. After the state transfer is done C sends out the new view to all members (including the new member) and user requests are now allowed again. If the coordinator fails, the group members will receive a failure notification from the communication subsystem and there will become a new coordinator, the next oldest member, and the view transfer will be restarted with the new coordinator.

Summary

The object group pattern can be used to provide fault-tolerant client/server systems through active replication, passive replication or multi-versioning. The pattern can be used to provide efficient group communication over any multicast channel through providing a high-level invariable interface to such functionality. Load sharing and scalability are also possible through splitting tasks on different members. This is why the object group design pattern is used in this system.

There are some drawbacks to the object group pattern such as requiring fault-tolerant name servers and putting a really high bar for the communication system. It needs realiable, totally ordered multicast of network messages, a group management protocol to ensure that object group members have consistent views on which objects are in the group, failure detection, and consistent propagation of failure notifications.

3.4 Controller

Behind the API we have the controller that handles all complexity with synchronization and replication of data. The API together with the controller contains the functionality of ensuring that the object put into the RLDS is stored properly and replicated to all recipients as well as ensuring updates/insertions on other members are propagated to this instance. The controller part handles the request from the API and handles the intergration and interface towards the in-memory store. Incoming objects from the API are put down through the GCM. When an object later is delivered from the GCM the object is stored in-memory and possibly on disk.

The initialize method of the RLDS interface starts the GCM implementation and creates/joins a given group. The method also registers message and group view listeners which can be used by client applications to get callback upon view changes in the group.

Upon putting an object into the RLDS the controller sends the message through the GCM and only stores objects that are delivered from the GCM. This means that all objects are put through the replication system and received/delivered through this to ensure that all members receive the same object, and in the same way.

When the GCM deliveres a group view change to the controller the RLDS determines if it is a new member and then the RLDS enters synchronization mode. If it is a rejoin from a member the members exchanges objects that are new since the group view index when the member left/crashed. When the synchronization is done the controller enters primary mode and works as normal.

3.4.1 Replication Model

The replication algorithm is based on the database replication algorithm presented in [1]. The algorithm is efficient and provably correct for database replication over partitionable networks that provides global persistent consistent order. Compared to two-phase commit protocols this algorithm can perform higher throughput (actions/second) as demonstrated in [1].

The system model consists of a set of nodes that each hold a complete copy of the database. The crash of any component running within a node will be detected and treated as a global node crash. A node may crash and may recover with its old identifier. The nodes communicate by exchanging messages through a group communication layer which provides reliable multicast messaging with total ordering guarantees. The communication layer also provides notifications to the replication system on node crash/recover and node voluntary join/part. This is detected through the communication sub system.

The conceptual algorithm is divided into four states; Prim, NonPrim, Exchange and Construct, as illustrated in Figure 3.8. In Prim state the node belongs to primary component. When a client submits a request it is multicast using the group communication to all the nodes in the component. In NonPrim state the node belongs to a partitioned group and requests are queued at the node. A node switches to the Exhange state upon a view change. All nodes in the view will exchange information to synchronize their knowledge. In the Construct state all nodes that synchronized in the Exchange state can attempt to install the primary component by querying all members with a create request. If all members response state will change to Prim state. If a view change happens before all responses are received the state changes to Exchange.

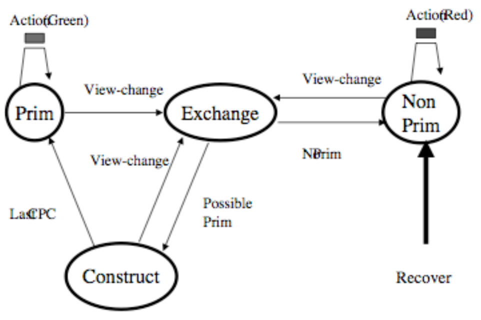


Figure 3.8: Conceptual Replication Algorithm

3.4.2 Replication Styles

There exists two major replication styles, passive and active replication [11].

In **passive replication** a single *primary* replica executes all invocations with one or more *secondary* replicas poised to take over from the primary if the primary fails. Periodically the state of the primary is retrieved and multicast to the group. Recovery involves electing a new primary, its initialization to the last checkpoint of the primary and the subsequent replay of logged messages to restore its state to the point just prior to the failure.

In **active replication** all of the replicas of an object are loaded into memory, and operate concurrently. Invocations and responses are multicast to all of the replicas, every server replica executes every invocation and returns a response. Thus, the occurence of a fault at one replica could be masked by the continued operation of other replicas. Active replication is desirable in real-time systems since the latency is the execution time of the fastest *live* replica. However it is more memory intensive and CPU intensive.

The one used in the system presented in this thesis is the active replication.

3.4.3 Replica Consistency

One assumption made on behalf of replication is that all replicas of an object should be indinstinguishable in function and behavior. All replicas must maintain a consistent state and handle invocations and results in an identical way. In short, Narasimhan [11] makes the follow criteria for replica consistency.

- Replicas must behave in a deterministic manner at all times. Two replicas for the same object that start from the same consistent state and performs the identical invocations are guaranteed to reach a consistent final state.
- Replicas must start operation from a consistent initial state. All replicas must have identical initial state at system startup. During recovery from a failure a new or recovering replica will start from the last checkpointed state and replay logged invocations and reponses to restore its state to the point where it can process new messages.
- Replicas must execute the same sequence of invocations. By assuring this and that all replicas start from the same consistent initial state, we guarantee that all updates to internal state occur in a consistent way across all replicas.
- Replicas must process each distinct invocation only once. In active replication all replicas execute every invocation and returne a response, resulting in duplicate results at the client. In passive replication a recovering primary replica might receive a repeat invocation of an invocation already processed by the failed primary. These duplicate messages must be detected and discarded.

The proof-of-concept system relates to these criterias in the following way. Replicas start out with the same empty state and performs the same actions, ensuring the same consistent state. If a replica crashes and recovers it will perfom the actions missing from its crashing state to the other replicas state. The underlying communication layer ensures that duplicate action is not delivered to the system making sure that invocations only are executed once.

Chapter 4

Evaluation and Testing

This chapter shows the evaluation of the proof-of-concept system and the test performed to demonstrate that the system performs as it should.

The tests are performed on two Solaris 10, Sun V240, 2GB ram, 2x1.3GHz UltraSPARC IIIi server machines located at the department of Computing Science, separated by gigabit (1Gbps) ethernet.

4.1 Evaluation

The evaluation and the profiling of the proof-of-concept system is performed with the help of *jiprof*, a profiling agent that logs everything you do in the Java Virtual Machine (JVM) and presents this in a text-document. When using a profiling agent to get a picture of the memory and processing footprint of an application it is important to note that the profiling agent poses some overhead on the system. The profiling agent I used, jiprof, creates an overhead of 100% which means that running the system with the profiling agent runs twice as long as without.

4.2 Testing

Apart from the JUnit test that have run during the development of the proof-of-concept system a few overall test cases have been created and executed to show that the system is working properly.

The first test is developed to show that the group communication works and the case is as follows. Create a group manager and run two group communication instances from two differnt computers, both connected to the internet. Connect these members to the group manager and have one of the members creating a group and have both members joining the group. After this, one member will send a large amount (10000) sequential strings to the group. That members as well as the other member will check that it receives all strings and in the right order. After this a crash is simulated for the group manager by “killing” the corresponding system process. Now the same large set of strings will be sent again and verified. After this a simulated crash is created for one of the members. The remaining member now sends a couple of messages to the group (with only itself as member). Now the “crashed” member is restarted and sends out a couple of messages to the group.

The second test has its purpose to show that the store works, both the memory store and the disk store. The test case starts out with the creation of a memory store with a disk store attached to it. After this 50000 Java Objects are put into the store and the program is closed. Now the store with disk store is initialized with the data files created previously. All 50000 objects are fetched from the store and verified. After this the store is cleared and all 50000 objects (now removed) are attempted to be fetched from the store, which will result in 50000 error messages.

The third test case test, the replication part and the overall function of the system. A group manager is started and two instances of the RLDS are started on two different computers both connected to the internet. Both members connect to the group manager and joins the group. After this one member put a couple of objects into the RLDS and the other member verifies that it has these objects in its own local store by fething the objects. After this, one of the members are simulated to crash and the remaining member stores a couple of more objects into the RLDS before the crashed member recovers. The crashed member will now recover (that is, start up again) and try to access the objects the other member has stored while the crashed member was down. Now the recovered member will store a couple of objects into the RLDS that the other member will retrieve.

Chapter 5

Conclusions

This report has shown the work of designing and creating a proof-of-concept system, a replicated local data store, to act as a generic common data store to be used by OMS and other applications through a given application programming interface. The system handles replication between instances separated by network/ethernet and recovery from certain host failures, including network failure and host crash.

5.1 Limitations and Future Work

Dispite the fact the proof-of-concept system works there still exist some limitations that leaves room for improvements. The proof-of-concept system has no support for primitive data types which means that even to put a simple primitive integer `int` into the system it has to be autoboxed into an `Integer` object to be accepted into the system. In a future version of the system it would speed up the system by supporting primitive data types. However this is coupled to the serializing method used and would mean that a custom serializing method would have to be created. This custom method would have other benefits as well. For example, if the serializing would be detached from the system different methods could be created which would be optimized for the current data types of the current instance.

One limitation of the proof-of-concept system is that it only handle host failures and treats all errors as such. This is fairly naive approach and could be developed and evaluated further. Possible it could also improve the perfomance of the system.

The only memory eviction policy implemented is the least recently used (LRU) policy and this is enough to provide a proof-of-concept system but in a real system more eviction policies would be desired.

The guarantees given in subsection 2.2.2 that the system should recover from disk under one minute or under 30 minutes with prepopulated chache is not guaranteed by the system. Tests have shown that it works with a large number of objects stored in the system but there is no limit on the maximum number of objects stored in the system.

Group Communication Module

The election algorithm chosen for the group communication is designed for synchronous systems and the proof-of-concept system is an asynchronous which means that the algorithm is not provable correct. The algorithm should be exchanged in a future version for e.g. the invitation algorithm or a ring based election algorithm.

The ordering part of the communication module only supports total ordering which gives no other guarantees but the same order at all member. In a future version this could be extended to use FIFO-total (first in first out) ordering which is also total ordered but messages sent from one member in order will be received by all other members in that order.

Controller

The replication model described and used in the system produces much overhead in terms of data traffic. For every message sent, that is every object stored, updated or removed from the proof-of-concept system approximately 50 messages are sent by the controller and group communication module. The replication model together with the group communication module is primary designed for two data centres and it does not scale well. The number of messages grows square, $O(n^2)$, with respect to number of hosts. This has a big negative effect on the performance of the proof-of-concept system and this means that the system most likely won't be used in a production environment. In a future version a different replicaion and communication model could be discussed to make the proof-of-concept system faster and more efficient.

Chapter 6

Acknowledgements

I would like to thank my external supervisour Marcus Nilsson at Nomura Sweden AB and my internal supervisour at the department of Computing Science Stephen J. Hegner. Thanks to all people who helped me with the report to make it better. A special thanks goes out to my wife Jannica for supporting me throughout the work on this thesis.

References

[1] Yair Amir and Ciprian Tutu. From total order to database replication. In *In Proc. of Int. Conf. on Distr. Comp. Systems (ICDCS)*, pages 494–503. IEEE, 2001.

[2] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, 1987.

[3] Kenneth P. Birman and Robert V. Renesse. *Reliable Distributed Computing with the ISIS Toolkit*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1994.

[4] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems (4th ed.): concepts and design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[5] Direct access trading. Web Page. Accessed Feb 13 2009, `http://en.wikipedia.org/wiki/Direct_access_trading`.

[6] Ehcache user guide. Web Page. Accessed May 13 2009, `http://ehcache.sourceforge.net/EhcacheUserGuide.html`.

[7] Brad Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004(124):5, August 2004.

[8] Hector Garcia-Molina. Elections in a distributed computing system. *IEEE Transactions on Computers*, C-13(1):48–59, January 1982.

[9] Silvano Maffeis. The object group design pattern. In *COOTS'96: Proceedings of the 2nd conference on USENIX Conference on Object-Oriented Technologies (COOTS)*, pages 12–12, Berkeley, CA, USA, 1996. USENIX Association.

[10] Microsoft project code named *velocity*. White Paper. Accessed May 13 2009, `http://msdn.microsoft.com/library/cc645013.aspx`.

[11] N. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Transparent consistent replication of java rmi objects. In *DOA '00: Proceedings of the International Symposium on Distributed Objects and Applications*, page 17, Washington, DC, USA, 2000. IEEE Computer Society.

[12] Order management system. Web Page. Accessed Feb 12 2009, `http://en.wikipedia.org/wiki/Order_Management_System`.

[13] A technical introduction to Terracotta. White Paper. Accessed May 13 2009, `http://terracotta.org/web/display/orgsite/TCWP+Introduction+to+Terracotta`.

Appendix A

User's Guide

This appendix is to be used as an aid for using the Application Programming Interface (API) to the system. The methods in the interface is described and in the end of this chapter an example is given.

A.1 Application Programming Interface

The `RLDS` interface is the application programming interface (API) towards the proof-of-concept system. The API supplies methods for insertion, deletion, updating and retrieving objects to and from the replicated data store. The API has the following methods:

`put` - Put a serializable object into the RLDS bound to a given key. `put(String key, Serializable o)`

`putSafe` - Put a serializable object into the RLDS bound to a given key. This method flushes the object to disk and to other group members ensuring that the object given as a parameter really will be stored. `putSafe(String key, Serializable o)`

`get` - Get the serializable object from the RLDS associated with the given key. `public Serializable get(String key)`

`delete` - Remove a serializable object from the RLDS associated with the given key. `delete(String key)`

`shutdown` - Shutdown method to ensure clean shutdowns. `shutdown()`

`flush` - Flushes data to disk, does nothing if no disk store is present. `flush()`

`initialize` - Connects the GCM to registry. `initialize(String groupName, String host, int port)`

A.1.1 Configuration

The `Configuration` object is sent to the implementation of the proof-of-concept system, the `RLDSImpl` class, at contruction with configurable values set. The configurable values are:

member name - A human readable name of this current instance. This is mainly to ease the debugging and development of the system.

path to data file - A path to the data file that will contain all the objects stored on disk.

path to index file - A path to the file that will contain the data store's index.

delete files on startup? - This boolean configuration value has an effect if there already exists files at the data and index paths. If this boolean value is set those files will be trucated at startup.

disk store thread sleep interval - The sleep interval of the disk store worker thread. This value affect how long the disk store will sleep and wait for data to be written to disk. Changing this value could optimize the performance of writing objects to disk storage. The value is in milliseconds.

disk store index write timeout - Similar to the disk store thread sleep interval. This value affect the maximum number of milliseconds between the data store's index is written to disk storage. Changing this value could optimize the performance of writing objects to disk storage.

controller timeout - This value determines how long time the group communication module (GCM) should wait before classifying an group member as failed. The value is entered in milliseconds.

A.1.2 Group Manager

The group manager is needed in the initial setup of a group and acts a nameserver that maps a group name to its members. The group manager is started through the `Server` class which takes one argument, the port number upon which to listen to incoming connections from prospecting group members.

A.1.3 Examples

Two examples follows. The first example, Listing A.1, is an example of creating an RLDS instance and accessing it through the API. The second example, Listing A.2, is an example of creating two RLDS instances, connecting them and accessing objects through both instances.

```
// Start the group manager
Server s = new Server(3003); // Port
s.start();

// Configuration
Configuration config = null;
config = new Configuration("member␣Foo", "index.dat", "data.dat");

// RLDS instance
RLDSImpl rlds = null;
rlds = new RLDSImpl(config);
// Connect to the group manager and
// create/join the group "grp1"
rlds.initialize("grp1", "localhost", 3003);

// Create a sample object and key
String key = new String("bar");
Serializable object = new String("Moose");

// Put the sample object into the store
rlds.put(key, object);

// Get the 'Moose' back from the store and print it
String o = (String)rlds.get(key);
System.out.println(o);

// Delete the 'Moose' from the store
rlds.delete(key);

// Trying to get the 'Moose' from store now
// would cause an exception to be raised
// rlds.get(key);
```

Listing A.1: Example of creating an RLDS instance and accessing it through the API

```
// Assuming that the group manager is started at some
// 'host' with port 3003

// -- At host A --
// RLDS instance A previously initialized
// RLDSImpl a = ...;

// Connect to the group manager and
// create/join the group "grp1"
a.initialize("grp1", "host", 3003);

// Create a sample object and key
String key = new String("bar");
Serializable object = new String("Moose");

// Put the sample object into the store
a.put(key, object);


// -- At host B --
// RLDS instance B previously initialized
// RLDSImpl b = ...;

// Connect to the group manager and
// create/join the group "grp1"
b.initialize("grp1", "host", 3003);

// Create a key for the 'Moose' object
String key = new String("bar");
// Get the 'Moose' back from the store and print it
String o = (String)b.get(key);
System.out.println(o);
// Notice that instance A added the object and
// instance B can access it

// Delete the 'Moose' from the store
b.delete(key);


// -- At host A --
// Trying to get the 'Moose' from store now
// would cause an exception to be raised
// because instance B deleted it
// a.get(key);
```

Listing A.2: Example of creating two RLDS instances and connecting them

Appendix B

The Bully Algorithm

The bully algorithm is a leader (or coordinator) election algorithm used in distributed computing systems to dynamically select a leader by an process identifier [8].

Each process starts up with an unique identifier that can be ordered and has a list of all members in the system. When a process P notices that the coordinator no longer is responding to requests process P initates an election as follows:

1. P sends an ELECTION message to all processes with higher ID.
2. If P gets no responses, it wins the election and broadcasts that P is the new leader.
3. If P gets a response from a process with higher ID, that process is now the leader.

A process can receive an ELECTION message at any time and if the process initiating the election has a lower identifier than the receiving process the receiving process replies an OK message and starts its own election (unless the receiving process is already running an election).

If a process crashes and recovers it will start an election. If this happens to the process with the highest identifier it will win and thus the “bully” will win.

As described above any process that receives an ELECTION message from a process with lower identifier will forward this ELECTION to its higher members and thereby ensuring that the process with highest identifier will always win.

In best case the process with next highest identifier will notify that the coordinator is down and only send out messages to lower processes telling them about their new leader. In the worst case the lowest identifier will notice the coordinator is down and all higher processes will run elections.

Appendix C

Source Code

The source code is available per request.