Robotic Devices as Interfaces to Web Services
by
H.J.T. van de Crommert

Supervisors:
Dr. Ir. T.A.C. Willemse (TU/e)
Dr. L. Holenderski (Philips Research)

Eindhoven, November 2007
Abstract

When creating applications that use devices to interact with web services, an application developer has to tackle several problems. Different communication protocols (TCP/IP, HTTP, XML-RPC) exist and each device or web service can add specific features or restrictions to these protocols. In this thesis, a platform is presented that tackles these problems, reducing the work of the application developer to define the application logic. The platform is based on the generalisation of devices and web services to reactive services, that can be connected much like dataflow process networks. The messages sent over these connections are scripts. Scripts provide a convenient way of communicating code, modifiable until the moment of evaluation. Apart from the development of this platform, a data replication protocol, usable for an essential part of the platform, has been modelled and verified using the mCRL2 tool set.
Acknowledgements

During the spring of 2006 I followed the course on Software Testing by Judi Romijn. Her enthusiasm and skill to explain complicated material in an understandable way triggered me to contact her to discuss potential graduation projects. Thanks to her connections, an interesting project at Philips Research was brought to my attention. Although Judi Romijn was unable to supervise my graduation project, she introduced me to the people who could.

My supervisor within Philips Research, Leszek Holenderski, helped me in the creation of a prototype and enlightened me with his visions on the world in general and the Netherlands in particular. The near inexhaustible number of facts, theories and opinions that he has got directly available surprises me still.

For the modelling and verification of a data replication protocol, my supervisor from the TU/e, Tim Willemse, proved to be invaluable. His questions about the protocol that I was modelling helped me in getting the details of the protocol, as well as the required abstraction level, right. After that, his knowledge on the internals of the tool set that I was using, as well as the assistance he offered on the theoretical foundation of which these tools are based was more than I could have hoped for.

I would like to thank them for their patience and support.

Henk van de Crommert
Eindhoven, October 2007
Contents

1 Introduction ................................................. 9
   1.1 Always online ........................................ 9
   1.2 The assignment .................................... 10
   1.3 The application .................................... 10
   1.4 The proposed architecture ......................... 11
   1.5 Thesis outline .................................... 11

2 The problem at hand ...................................... 13

3 Typically known solutions ................................. 15
   3.1 General definitions ................................ 15
   3.2 Message Passing ................................... 16
   3.3 The RPC paradigm .................................. 17
   3.4 In conclusion ..................................... 20

4 The proposed architecture for reactive services ...... 21
   4.1 Essential ideas ..................................... 21
   4.2 Architecture ....................................... 23
   4.3 Socket layer ....................................... 23
   4.4 Service layer ...................................... 25
   4.5 Application layer .................................. 27
   4.6 Correspondence to problem statement ............... 27

5 Application ................................................. 29
   5.1 Join me in SecondLife! ............................. 29
   5.2 The setup ......................................... 29
   5.3 Creating the application ........................... 30

6 Verifying data replication ............................... 33
   6.1 Data consistency between distributed hosts ....... 33
   6.2 A serializable protocol ............................ 34
   6.3 Requirements of the system ....................... 35
   6.4 Model checking tools .............................. 36
   6.5 The actions of the model ......................... 39
   6.6 Formalising the requirements ..................... 43
   6.7 Modelling the protocol ........................... 46
   6.8 Evaluation of the requirements ................... 53
Chapter 1

Introduction

We live in a world in which the information that can be found on the world wide web is no longer available only to those people that sit behind their desktop computer. Instead, devices that grant access to this information from any given place and time like mobile phones are becoming more popular and affordable.

Within the BRAINS project at Philips Research this development was considered to be interesting. The project group’s main focus lies on Human-Robot Interaction and as part of the Connected Consumer Electronics group, there was a natural interest in using the robotic devices they developed to interact with web services. Since the development of these type of applications is more complicated than is desirable, a platform should be designed that hides these complications for the application developer.

This chapter will start by introducing the vision within the project team, creating consensus on the context of the architecture. There will be a short overview of the difficulties that the architecture should address. After the problem has been properly introduced, the proposed solution and the way in which it can be used is briefly described.

1.1 Always online

During the last ten to fifteen years, the Internet and the way in which it is used changed drastically. Starting out as a gigantic library with static information that users anonymously browse through, it became more social with the rise of community driven websites and online chat boxes. With the increasing popularity of virtual worlds that run on the Internet like Second Life and World of Warcraft, with constant activity, one could spend the entire day on the Internet without ever being bored.

In reality, it is not possible nor desirable to be positioned behind a desktop computer the entire day. To prevent the user from missing interesting online events, while still enabling the user to leave his desktop computer every once in a while, the BRAINS project team envisions the availability of robotic devices that can be placed at strategic positions throughout the house. These robotic devices are to be designed in such a way that they present an interface for the online service, ranging from a small light bulb that signals the user when a new email is received or a telephone to communicate with your friends in Second Life.


1.2 The assignment

As an application developer, creating applications with distributed components that are
owned by different vendors is cumbersome at least. There are various protocols available
for interaction with web services and the same holds for robotic devices. This situation is not
likely to change over the next few years, despite the effort put in standardising protocols used
for remote computations such as CORBA. In order to cope with this, application developers
need to study various protocols as well as crafting the application logic. As an effect of this,
the time needed to develop new applications is much larger than desirable.

To simplify the work of the application developer, an architecture has to be provided that
abstracts over these different protocols and allows the application developer to focus on the
application logic itself.

1.3 The application

During the project, a single application was used to generalise the architecture from, using
the iCat as a robotic device and Second Life as a web service. The iCat is a robotic device
and a result of the work done within the BRAINS project. The device is used to investigate
human-robot interaction, with the emphasis on simulated emotions. The robot is shaped as
an immovable yellow cat and is capable of expressing different facial emotions as shown in
figure 1.1.

![The iCat, showing several emotions](image)

Figure 1.1: The iCat, showing several emotions

Second Life is a virtual world in which people control a virtual avatar. This virtual world
is entirely created by the inhabitants that shape the environment, modify their appearance or
start businesses by selling objects and appearances that they have created. The main activity,
 apart from creating the world, is the social life within Second Life. Second Life has been used
within research experiments and companies are investigating the business potential of virtual
worlds.

The following application was considered to be representative for the application area:

**iCat and Second Life**  Suppose that you are playing Second Life. After playing the game
for some time, you start missing your best friend and you wish to invite him. After selecting
him in your in-game address book, a message is send to your friend’s iCat. The iCat starts looking around and informs your friend that you are online and would like him to join you. By pressing the iCats left paw, your friend informs you that he will soon be online.

1.4 The proposed architecture

In the architecture proposed in this thesis, a layered approach is taken. Three layers are defined, separating the functionality into communication, component specifics and application logic. This layered approach is common practise for most distributed computation systems, but some twists are added.

The communication layer is responsible for communication of data, communicated by message passing. The messages passed are scripts, which is plain text data until this data is evaluated by an interpreter.

The service layer abstracts away all protocol and service specifics, apart from its functionality. Following the way in which robotic devices are perceived, all services are considered to be reactive. This means that the services can autonomously trigger events, as well as receive input from the user.

At the application layer level, application developers define the application logic that connects the components as if they are part of a process network. The connections are defined via small pieces of code, allowing the application developer to translate between service message formats.

For the localisation of services on the service layer, a registry is used. This registry may consist of several servers that store the same data. This data must be kept consistent, which can be done by using data replication protocols. The consistency is a vital property of the registry, directly influencing the operation of applications. Because it is such an essential element, guarantees on the protocol behaviour are valuable. A formal verification of a replication algorithm gives the best guarantees and is given in this thesis.

1.5 Thesis outline

In this thesis, the architecture that was shortly introduced in this chapter will be described in more detail. The details of the assignment will be given in Chapter 2. In chapter 3, two different paradigms that are used by distributed computation systems are presented, such that the reader will get some knowledge on existing systems. In Chapter 4, the architecture of the system and the separation into three layers will be described. The way in which this architecture simplifies the work of the application developer is described by an example application in Chapter 5. The formal verification that is mentioned in section 1.4 is given in Chapter 6.
Chapter 2

The problem at hand

In section 1.2, a short introduction of the assignment was given. In this chapter, a more detailed description of the expected environment in which the platform is expected to be used is given. By defining the environment in an early stage, later chapters of this thesis do not have to consider options that fall outside the scope of the proposed architecture.

The assignment itself consists of the design and prototyping of an architecture that enables an application developer to create applications that involve multiple robotic devices or web services. The desire for such an architecture came from the observation that it required quite some effort to develop applications that had this behaviour, even when all components are provided by the same company. To give an indication of the varieties that are possible in the architecture that should be designed here, consider figure 2.1.

![Figure 2.1: The different kind of robotic devices and web services and the way in which they can communicate.](image)

In this figure, the iCat and Nabaztag represent two different robot types. The iCat allows communication in both directions. It communicates via DML (Dynamic Modelling Language), which is an internal framework on top of PVM (Parallel Virtual Machines see Chapter 3.3.1).

The Nabaztag is a white rabbit-shaped device, which can play sounds and animate its ears. All messages addressed to the nabaztag are delivered to the device when the device polls its owners community web page. On this community page (named My Nabaztag) all data sent to the Nabaztag is temporarily stored, similar to the central storage of e-mail messages.

Considering all possible types of robotic devices and web services leads to a wide range of processes that can be interconnected. Each of these processes defines its own functionality and can use any communication protocol. To complicate matters, some of these devices might...
be programmable, such as SecondLife and the iCat, which makes the number of available functions limited only by the creativity of its users.

Another challenge is the localisation and addressing of robotic devices and web services. The web services are available via the Internet and under a well known address. This is not necessarily true for the robotic devices. Another variation is the usage of a single well known interface to address several robotic devices or services, as is done within Second Life.

In the remainder of this chapter, a short introduction on the most critical requirements of the system will be given.

**Platform and software restrictions** Although an architecture should not depend on any particular technology, it should be stressed that in the architecture that will be described in the remainder of this thesis, no operating system or language dependent technology may be used in the design of the platform.

During the creation of the prototype it is allowed to use operating system or language dependent solutions, under the assumption that equivalent functionality can be obtained in different programming languages and operating systems.

**Communication type** The popularity of web sites providing streaming audio or video is increasing together with the available bandwidth of communication networks. Applications that combine robotic devices and these kind of applications are very appealing, but they come with more stringent timing requirements than the type of applications that are represented by the example application of Chapter 5. Using this application and assumptions on future projects as a guidance, the platform is required to support the communication of data packets.

**Network availability** The robotic devices and web services can be located all over the world. This potentially introduces problems when these processes are positioned on different networks, caused by an intermediate *Network Address Translation* (NAT) router. A NAT router enables multiple computers to access the Internet, while consuming only one IP-address. It has been introduced several years ago as a temporary solution because IPv4 was running out of available addresses and IPv6 was not yet available. It enables the reuse of addresses, which solves the problems caused by having only a limited number of addresses, but it complicates bidirectional initialisation of communication as addresses no longer have to identify unique processes. With a NAT router in between, it is rather complicated to initiate a communication from any arbitrary computer outside the local network to a computer within [32].

There are more problematic situations imaginable, and most of them can be solved by selecting the appropriate communication protocol. In the remainder of the thesis it is assumed that processes can be connected and that connections are reliable.

**Scaling** In the platform, multiple web services and robotic devices must be connected and controlled by a single application. The number of simultaneously running applications and connected devices may be limited only by hardware restrictions.
Chapter 3

Typically known solutions

The requirements (section 2) imply that inter-process communication is an important part of the desired functionality. According to the application (section 1.3), communicated data will trigger some predefined behaviour at the side of the receiving process.

When communication over computer networks is involved, there are two major paradigms to take into consideration. The first paradigm is *message passing*, where components send each other packages with data. In the basic *remote procedure call* (RPC) paradigm as well as the *Object Oriented* variants, a component invokes a procedure with appropriate parameters on a remote machine and receives the result of this procedure call as the returned value. In the following sections, an overview of these paradigms and some example libraries are introduced. It is worthwhile to invest some time in reading about previous solutions, since knowing the advantages and disadvantages of these systems allows for better design of a new solution or selecting an existing one that best matches the applications requirements or environment.

3.1 General definitions

There are some variations in communicating messages that are available for both paradigms that will be introduced later in this chapter. Communication between processes can be either synchronous or asynchronous and blocking or non-blocking. In case of synchronous communication, the action returns when the other side of the communication has accepted the connection and received the message. In case of asynchronous communication, the actual connecting and transferring of data is deferred to another process. This additional process handles the communication of data and returns control as soon as the data has been delivered to the additional process. This implies that the behaviour of the synchronous communication action is *blocking* and causes the process to halt when the remote process is not ready to receive a message. Asynchronous communication does not suffer from this behaviour and is therefore non-blocking.

The two concepts ((a)synchronous and (non-)blocking) are not symmetrical. An action might write data to a buffer, thus being asynchronous, but may only return control when the data has successfully been transmitted to the buffer of the remote process, making it a blocking call.
3.2 Message Passing

In message passing, processes send each other messages, which may be acknowledged after they have been successfully received. A message consists of data, formatted as specified by the sending process. This data can exist of signal values or function calls, but any other type of data is also valid. The process sending the data makes no assumption on the way in which the information is handled and is not directly interested in the results, if any.

This paradigm follows the intuitive idea of communication between different processes, without transfer of control. After sending a message, a process forgets all about it and continues normal execution. Systems based on this paradigm benefit from an event-driven programming style, constantly awaiting new messages and handling them as they arrive. Event-driven programming tends to be less complex and results in more robust software when compared to threaded programming [8].

3.2.1 Implementations of Message Passing

Message passing is common used in parallel computing, where it is used for the coordination of tasks and communication of data. Two frequently used tools for parallel computing are Parallel Virtual Machines (PVM) and Message Passing Interface (MPI), which will be introduced in this section.

Parallel Virtual Machine

The major achievement of the Parallel Virtual Machine [12] software tool is the abstraction over a heterogeneous set of computers to one distributed parallel processor. Brought into existence as a by-product of research at the Oak Ridge National Laboratory in 1989, it has been developed further ever since and is used mostly in the scientific world.

PVM operates by running one or more daemons on every host that is connected. These daemon processes handle message passing, authentication, process control and provide fault detection. Of the daemon processes that reside on the same host, one is designated to be the master process and this process is the only daemon that can create new daemons, disconnect hosts or reconfigure the system. Also, a set of hosts that are available is kept synchronised between the hosts that make up a virtual machine. From this set of hosts, a subset can be selected to perform specific tasks.

In parallel systems, processes are optimised by either dividing separate tasks that are independent of each other to different processors (functional parallelism) or by dividing the data to separate processors (data parallelism, or Single-Program Multiple-Data). Both of these divisions can be realised by the use of PVM, and the library contains primitives to make this division more convenient.

Within the virtual machine, tasks have a unique task identifier which can be passed among hosts in order to identify the context that belongs to the specific task. By storing variables and user defined functionality in the context, different tasks do not interfere. A user can observe and manipulate its task identifier, making it impossible to have two distinct PVMs communicate since this could have the effect that identifiers are no longer unique [16].

The message passing handled by the daemon takes place in a non-blocking way, which means that after passing a message to the network, the calling process continues normal execution. The communication itself takes place via either the TCP or UDP protocol. In-order and reliable delivery are guaranteed by the daemon.
3.3. THE RPC PARADIGM

Message Passing Interface

The message passing interface (MPI) has been designed by a forum of developers, library writers and end users. It has been compared to PVM several times [16], but the two do not solve the same problem. Where PVM creates a virtual machine, MPI has been designed as a library specification for message passing, not regarding the creation or management of new processes [15]. The heterogeneity that is part of the PVM is not required in real-world implementations of the MPI specification, although the specification itself supports heterogeneous computing by providing its own data types.

In MPI a context and a group of processes, referred to by their rank in the group, are stored in a communicator. An inter-communicator combines the context and processes of two communicators. If a process wants to communicate with a process in a remote communicator, it uses the inter-communicator which makes the reference to the process at the specified rank in the remote communicator. So contrary to PVM, communication between two groups of processes is possible.

3.3 The RPC paradigm

The basic functionality of a procedure call consists of directing the processor to a specific memory location where the next instruction to be executed is placed. Parameters can be given to a procedure call to influence the outcome of the execution. The redirecting of the processor implies a transfer of control, and the calling process suspends while waiting for the result of the procedure [5].

The Remote Procedure Call model extends the basic procedure call model by making procedures callable that run under different threads, possibly controlled by other processes on remote machines. The success of an RPC system depends on the level of transparency of the system. This level of transparency depends on the level to which local and remote procedure calls are syntactically and semantically equivalent. The level of transparency is influenced by the performance and robustness of the system, which is not identical in the local and remote case due to the negative side-effects of network communication (such as delays and unreliable channels) [34].

In the standard model of an RPC system [29] (see figure 3.1), every process includes the RPC run time support. This run time support communicates with a stub that presents an interface for calling remote procedures to the application developer. To serve procedure calls from other threads, all accessible application procedures are made available through the stub. The stub provides the marshalling of data formats and conversion of procedure calls, required for communication between programming languages and operating systems. Stubs can be generated by providing an interface file, specifying the imported and exported procedures. This interface file has to be written in an Interface Definition Language (IDL).

Contrary to normal procedure calls, the transfer of control followed by a procedure call can be undesirable when that procedure call is remote. As such, RPCs can be synchronous or asynchronous [37]. In an asynchronous RPC the process continues execution after the procedure call has been made. When the result of this RPC becomes available, the process returns to the part of the program that invoked this RPC. To find the corresponding program part, identifiers are attached to every non-blocking RPC in the following format (where \([a]\) indicates that \(a\) is optional):
Programming with the RPC paradigm is similar to procedural programming on single processor machines. A thread is created for every RPC and preemption of a thread ensures that local computations can still take place during the communication and computation of a remote procedure call. This programming style is widely known and used, but has some drawbacks as far as resource usage and resource sharing are concerned [8].

### 3.3.1 Implementations of RPC

Since the beginning of the nineties, several vendors created their own implementations and several attempts were made to introduce standards. In this chapter, some of the major RPC systems will be introduced, as well as a quick overview of the strong and weaker points of each variant. The section starts with CORBA, which was introduced as the ultimate solution to the inter-machine communication problem, but failed in this respect for a number of reasons. Several alternatives for CORBA were introduced for Java programmers (EJB, Spring) or based on the XML standard. In the remainder of the chapter, some information will be given on the JAVA and XML-based solutions.

**CORBA**

The *Common Object Request Broker Architecture* model was one of the first to support the RPC paradigm in a generic way. It was designed and promoted by a consortium which is called the *Object Management Group* (OMG), consisting of eleven different companies. The model used is similar to the one given in figure 3.1, where the RPC run time is renamed to *CORBA Orb*. Nowadays, CORBA is mostly used in company networks that are shielded from the Internet and for real time or embedded development [17].

Despite its support by several vendors, CORBA has never become the glorious standard it was designated to be [17]. This was partly due to the way in which the consortium was
functioning. Instead of defining a small and concise set of functionality and verify this set by developing an implementation, adding functionality only if required, a less incremental approach was taken. Within the consortium, the companies that wanted to use the technology provided a list of requirements for the system. This list was given to the companies with the knowledge to develop this functionality, which all created their own set of functions. Creating the specification of CORBA was done by taking the union of all provided functions, leading to a specification that was complex and contained contradicting functionality [17].

Having a specification available does not guarantee interoperability between implementations. This was especially true in the CORBA case, where vendors had great difficulty to implement their system according to the specification. Additionally, several vendors added functionality to their implementation to distinguish their implementation from their competitors.

The interface of a CORBA component is specified via an IDL file. This enables other components to use functionality provided by the component, but also causes recompilation of all components involved if the functionality would ever be modified [7]. Differences in programming languages include differences in type definitions, which require conversions between different data types or taking the smallest common set of types [20].

Summarised, the problems around CORBA mostly relate to incompatible implementations and different restrictions on data types. Add a steep learning curve due to the complex specification to the list of problems and the system was deemed to fail. This indeed has happened and several of the original members of the OMG have already abandoned CORBA, as well as most of the users of the system.

Java

One of the main ideas behind Java is the ‘write once, run everywhere’ concept. This is realised by compiling Java programs into byte code, which can be interpreted by any Java Virtual Machine (JVM). The type system that comes with the JVM effectively solves the type conversion problem mentioned earlier.

Enterprise JavaBeans The Enterprise JavaBean (EJB) is the official approach of Sun Microsystems to component-based software engineering. By using Java Remote Method Invocation (RMI), the object oriented approach to the RPC paradigm, methods on different machines can be accessed. EJB is developed to standardise the way in which the general back-end functionality of application servers are developed.

In the initial versions of EJB, EJB components had to implement specific interfaces and were tightly coupled to the framework. Although separate from the code, concerns like security and persistence needed to be taken into account when creating the business logic. After creation, the entire business logic would have to be reevaluated whenever an update of the framework became available [31]. Similar to CORBA, the EJB framework suffered from complex interfaces which caused programmers to avoid the framework.

Observing these problems, several alternatives to EJB were presented by the open source community. By using POJOs (Plain Old Java Objects) instead of the complex EJB components, concerns are separated better in frameworks like Spring than in the EJB approach and interfaces are more clear and less reluctant to change [6].

The developers of EJB recognised the shortcomings of their system and changed their approach to a model similar to that of the open source community.
Spring framework  The Spring framework [19] was created with the disadvantages of EJB in mind. Reducing the coupling between components, making the usage of the framework less apparent in the development of the business logic as was the case with EJB, developed applications are hardly influenced by changes within the framework. The Spring framework itself consists of several smaller frameworks, which can be used either individually or to compose larger applications. Some of the frameworks provided provide aspect-oriented programming, transaction management and remote access via protocols as CORBA, RMI and SOAP.

SOAP/XML-RPC

Both of the previous solutions to a certain extent restrict the application developer in the choice of programming language and components in order to provide the communication of messages. SOAP differs from this in the respect that it only specifies the messaging format, having higher layers define the required functionality. Communication goes via standard HTTP ports, enabling SOAP messages to pass through firewalls more easily than CORBA or Java RMI.

The message format specified by SOAP is based on XML, a text based format that allows for the hierarchical structuring of data. The usage of XML makes messages human readable and the popularity of the language has led to a wide range of both commercial and freely available tools to operate on XML messages. A disadvantage of this message format is that it is verbose and that the parsing of messages is a time consuming process.

The protocol from which SOAP has evolved is called XML-RPC. This protocol defines a small number of data types and commands, making it a lightweight and comprehensible alternative for SOAP and both protocols are still in use. When comparing a program that uses the generic XML-RPC protocol with a task-specific program written in Java, the XML-RPC program turns out to be slightly less complex to develop but requires up to a magnitude more network and computation resources due to the encoding and decoding of XML [1].

Both XML-RPC and SOAP are based on the client-server model. In practise, a bidirectional communication might be convenient. For this purpose, the JSON-RPC extends XML-RPC with the option to connect over TCP/IP sockets instead of HTTP. Connections via TCP/IP can be used in two directions, allowing for more responsive applications.

3.4  In conclusion

Although presented as two different paradigms in this chapter, the two communication paradigms are equally powerful. Asynchronous RPC can be considered to be a layer on top of message passing, guaranteeing a message returned for every message received [24]. The choice for either paradigm heavily influences the style in which programs are written, even though there are few arguments for selecting strictly one paradigm.

It can also be concluded that having a group of several vendors implementing and using the standard is not a guarantee for success. Both the CORBA and the EJB approach indicate this. They failed to provide the developer with an interface that was rich enough to make interesting applications yet sufficiently small to be fully understood. It further simplifies the system when using a message format that provides minimal expressiveness as the POJOs in Java or is human readable like XML-RPC and SOAP.
Chapter 4

The proposed architecture for reactive services

The solutions presented in chapter 3.3.1 give an overview of the current state of inter-process communication protocols. Time has shown that none of them was the ultimate solution, although some of them claimed to be, but all serve their purpose. It is not immediately clear how using any of these systems would assist the application developer in creating an application to connect robotic devices and web services that all use their own protocols.

In this chapter, a layered architecture that actually simplifies the work of the application developer is proposed and described. Starting with introducing some technologies that were used as a basis of the proposed architecture and that are required to appreciate the design choices made, the chapter is followed by a per-layer introduction of functionality and the corresponding interface of that layer.

4.1 Essential ideas

Several technologies are used at the basis of the proposed architecture. These technologies are introduced in the following section. The section on scripting languages was based on information found in [36], which provides a chapter on this programming paradigm.

4.1.1 Scripting languages

System programming languages are used by programmers to produce a binary file that can be executed on a system. There is a wide range of different programming languages and domain-specific compilers are made to simplify the programmers work and further increase the quality of the resulting binary.

During compilation of source code, translating it from human readable program code to hardware instructions, several checks can be performed on the code in order to guarantee desired properties of a program. Since these conditions are verified during the compilation, no further checks during execution are required.

At times, a software engineer can encounter a problem that can be solved by orchestrating different existing binaries. For this goal, writing an additional application in a programming language is possible, but not ideal. The combination of strong typing and compile-time verification of safety properties is lacking in flexibility. An alternative way for the orchestration
CHAPTER 4. THE PROPOSED ARCHITECTURE FOR REACTIVE SERVICES

The use of scripting languages is the use of scripting languages. Scripting languages do not use their resources as efficient as program code due to the need of additional checks during run-time. This drawback is accepted, under the condition that the vast majority of the resources will be consumed by the components and not by the glue.

Programs written in a scripting language are not compiled, but make use of an interpreter, which evaluates the script line by line. This reduces the development time and enables easier modifications later on, since the source is directly available. All type and boundary checks take place during run-time. This leads to a highly flexible implementation as the type of parameters and the boundaries of an array can be determined during run-time. With dynamic typing, a single function can be sufficient to handle parameters of different types, without requiring a solution like C++ templates.

The interpreters that are used have been improved in the last few years. Scripting languages are used more to completely implement applications instead of glueing existing components. Several provide a high level functionality, allowing complex applications to be rapidly developed [30].

String manipulation with regular expressions

Matching strings and manipulating them are common operations in scripting languages. An approach taken by most scripting languages is the use of regular expressions, which provide the functionality to compare a string with an expression containing parameters. This is a convenient way to get information from a string, since it does not involve any additional string parsing to be programmed by the application developer.

High-level functionality

Scripting languages, in contrast with ‘normal’ programming languages, are designed with existing components in mind. This allows a higher level of abstraction, especially for application-specific scripting languages. This reduces the amount of code needed to create an application program [30]. Additionally, scripting languages make few assumptions on the underlying hardware platform. This allows scripts to be highly portable between platforms, as all hardware specifics are handled by the interpreter that is used to evaluate the script.

4.1.2 Dataflow Process Networks

In a data process network [26] or coloured Petri net [18], the inputs and outputs of different computations can be connected. This results in a graph, where the nodes are representing the computations and the arcs are data streams. This notation is standard in signal processing, where visual tools are an aid in creating the graph.

The ability to connect inputs and outputs of nodes visually by means of data streams helps in making the system understandable. Allowing nodes to be dataflow process networks themselves enables the hierarchical structuring of functionality. This is similar to the way in which this is done in conventional programming languages with procedures and components.

In order to determine the firing order of processes in a process graph, several execution models are available. None of these execution models is prescribed by the conceptual model behind dataflow process networks or coloured petri nets. By selecting an appropriate execution model the system may configure the graph on the fly or provide the most efficient firing sequence [26].
4.2 Architecture

Communication protocols are often designed in layers, inspired by the division of functionality in the OSI protocol stack. Each layer uses functionality provided by the layer underneath it and provides an interface to its abstracted functionality to the layer that lies on top of it.

By dividing the functionality required in the proposed architecture into three distinguishable layers, the functionality provided can be specified to the developers needs. Layering and encapsulation increases productivity and reduces the chance of making programming errors [38]. The following division of functionality is suggested, where the lowest layer is mentioned first:

- The socket layer for inter-process communication.
- The service layer for controlling robotic devices and web services.
- The application layer for defining the application logic.

With these layers, it is possible to create a network of communicating processes like the one presented in figure 4.1.

![Figure 4.1: The basic model of the architecture](image)

The remainder of this chapter will elaborate further on the details of each layer, adding interesting features in each layer. In order to improve readability, the differentiation between robotic devices and web services will be omitted where possible. Instead of this differentiation, the term service will be used where both are applicable.

4.3 Socket layer

The socket layer is the lowest level in the architecture and responsible for communication between services. In this section, the essentials of the socket layer will be presented.

4.3.1 Message format

The platform should support a message format that is framework, programming language and platform independent. This requirement is satisfied easily when using a text based format, for
which the marshalling between systems is straightforward. A commonly used text based data format is XML, which allows the structuring of plain text as a tree. Additionally, semantics can be added to elements of the XML tree, making elements optional and their order of the elements within sub parts of the tree arbitrary.

Combining several procedure calls within a single XML message is possible. If the process receiving this message is capable of executing the combined procedure calls in the same order as intended by the sending process, this allows a reduction in network load and response times. It is even possible to extend the functionality of XML and use the result of a previous procedure call as an input parameter to another procedure call.

Without strictly specifying an XML message format however, none of these features is available. Instead, a scripting language can be chosen to specify the message format. Some benefits of scripting languages have already been described in Section 4.1.1. Additionally, scripting languages allow the creation, inspection and evaluation of fresh or received scripts during run time. Messages can be communicated between any two components, making the scripts similar to mobile agents. The main difference that prevents the use of the term “mobile agent” with respect to the scripts is the lack of adaptability [25].

4.3.2 The model

Choosing scripts as a communication format requires the availability of an interpreter. This interpreter evaluates received messages and provides a mapping between the high level application description in scripting languages to platform specific hardware instructions. The model is shown in figure 4.2.

The interpreter keeps track of all available procedures. Following the Safe-Tcl security model [27], all potentially dangerous operations are unavailable or have restricted functionality. Resource usage can be limited for every interpreter instance. For every connection that is established an interpreter instance, child to the current interpreter, is created. This child interpreter serves as a sandbox for scripts received over this connection. Communication between sandboxes is only possible in a top-down fashion, from parent to child. Exceptions to this rule can be made via the use of aliases. Aliases allow a parent interpreter to declare procedures that are available to a child interpreter.

Aliases allow remote processes to address functionality that is present in the service layer. This security model, consisting of sandboxes and restricted access to potentially critical func-
4.4. SERVICE LAYER

tionality, allows the usage of scripts as a message format. It addresses the risks that are normally involved when communicating executable code in an convenient way.

Without proper authentication and encryption, it is impossible to determine individual privileges for remote processes. Authentication and encryption can be added by using the Transport Layer Security (TLS) protocol, allowing a more sophisticated security model. From Java 1.2, java applets are subject to a sophisticated model. Some of the advantages and potential problems in creating it have been described by Gritzalis et al. [13].

4.3.3 Communication paradigm

In Chapter 3.3.1, an introduction to message passing and RPC has been given. The application layer should provide at least one of these two paradigms, and might even include both.

Both paradigms can be used to implement the other and as such, it is not required to include both. However, passing scripts as messages poses two interesting possibilities that advocate the use of both paradigms. A script can either terminate with or without a result. In case of the former, the remote procedure call paradigm is more convenient. In case of the latter, message passing is the most appropriate solution.

In the socket layer, supporting both paradigms is straightforward. A continuous connection between two processes is established with TCP/IP or an equivalent protocol. A protocol is considered to be equivalent if it supports reliable and in-order delivery of packets between two end points and packets are received at most once. Over this connection, the message passing paradigm is used. Messages are bounded by a new line character and are used to reconstruct the received script in memory. After a complete script has been communicated, it must be followed by either a "MP" or "RPC" message. When this final message has been received, the complete script is retrieved from memory and evaluated in one atomic action. The result of this evaluation is returned to the sending process if and only if the RPC message has been received.

Every connection has its own interpreter. This interpreter runs in a single thread and it is the responsibility of the sending process to properly structure scripts.

4.4 Service layer

The service layer is placed on top of the socket layer. It consists of services which can be located anywhere on the Internet. In the service layer, all service specifics not directly relevant to the application writer, such as communication protocols and various communication models, are abstracted away.

4.4.1 The model

The service layer abstracts over all protocol specifics and the physical location of a service. The layer controls a service that may be local but may also be remote. Following the naming of robotic devices, it can be stimulated via its inputs and can communicate data via the outputs. Exact input and output values depend on the service itself.

When created as a layer in the platform, it looks as shown in figure 4.3. The term reactive service will be explained in the next section.
4.4.2 Reactive behaviour

The applications that benefit from this platform connect robotic devices to web services. Using this connection, applications can call procedures on the robotic device or web service. When regarding robotic devices, this remote control is not the only way in which the robotic device can be manipulated. Human interaction has proven to be an important factor in the application logic and must be communicated to the application layer.

To communicate this interaction to connected processes, several options can be considered. First, the application can periodically poll for the current status. Secondly, the application can subscribe for status messages following the publish-subscribe design pattern. Finally, a dedicated connection can be established for the communication of status messages.

It is worth noticing that web services in general and virtual worlds in particular may constantly change their current status. Similar to robotic devices, this information is interesting for the application logic and should be communicated.

Summarised, both the robotic devices and web services require bidirectional communication with the application that controls them. As stated earlier, this functionality can be implemented via the RPC or Message Passing paradigm. Doing so from the application layer would complicate the application logic unnecessarily, such that this should be handled in the service layer instead. The concept of a reactive service is proposed.

In a reactive service, interaction is split into inputs and outputs. Inputs and outputs are one-directional and communicate via message passing. A device is called reactive when it tends to send output values after being stimulated on its inputs. Inputs can be triggered both from an application and from other sources, such as a human user or the robotic device.

This approach is also taken in dataflow process networks and coloured Petri nets. The different components are represented as nodes in the graph and applications can be created by specifying the relation between inputs and outputs of different components.

4.4.3 Registering a service

The services may be located anywhere on the Internet. The localisation and establishment of a connection is hidden inside the service layer. To help developers in the creation of service layer implementation for their services, a single service should be provided up front. This service, named Registry, is a web service that resides on a well known address that is publicly available. A service that is available for remote connections publishes his address information via the registry. If the service layer wants to connect to a specific service, it can request the address information of this service via the registry. A service that has published
its availability via the registry must periodically update this information, ensuring that the stored information is as actual as possible.

4.5 Application layer

On top of the service layer and the socket layer lays the application layer. This layer is only present where application logic is defined, according to the architecture presented in figure 4.1.

4.5.1 The model

The application layer is shown in figure 4.4. The internal blocks, labelled $S_i$ for $1 \leq i \leq n$ define the desired behaviour. This behaviour is defined in terms of inputs and outputs. Note that the outputs of all connected services are perceived as the inputs of the application layer and the outputs of the application layer are connected to inputs of the connected services.

![Figure 4.4: The application layer of the architecture](image)

Applications written in the application layer can control the inputs of connected services. They are also informed whenever a new output is generated by any of the connected services. An application can specify a behaviour that must be performed whenever a certain output value is received. In this way, a specific output value can be translated to be used as an input value to other connected services. This is shown in figure 4.4.

As can be seen from figure 4.5, not all inputs or outputs have to be connected in the application layer. It might be more useful to have an internal event, such as a timer, trigger the inputs of a service. If a received output value is not required by the system, it may safely be ignored.

Taking a more hierarchical approach, an application by itself describes a reactive service. The inputs and outputs that have not been handled by the application logic can be controlled by other applications.

4.6 Correspondence to problem statement

The architecture presented in this chapter matches the criteria stated in Chapter 2. Communication takes place via TCP/IP, which is available for every programming language and platform, but this is hidden inside the socket layer and can thus be freely changed with a more appropriate protocol if required.

The data format is text based. This makes marshalling straightforward and allows human-readable communication. Requiring that the messages themselves are scripts that can be interpreted allows the structuring of data and procedure calls in a similar way as can be achieved
Figure 4.5: The logic within the application layer allows reactive services to be connected with XML, but with much less effort. Interpreters for scripting languages are available on every platform and can be used directly from other programming languages. On platforms where the resources used by an interpreter are too high, simple string parsing is sufficient to provide the basic functionality required.

The applications can be stored either centralised or distributed. In the centralised case, the responsibility for maintaining the hardware and providing sufficient network capacity is assigned to a single company. It allows this company to charge users of their resources and it is possible to create a community around the architecture and the applications. In the decentralised case, users can customise the behaviour of their own devices locally and independent of other processes.

Application logic can be represented as a graph with flexible vertices. Services are created independently of other services and the application logic provides the translation between inputs and outputs. The platform allows and supports the perception of applications as reactive services by themselves.
Chapter 5

Application

The major goal of the platform was to simplify the work of the application developer. By hiding the protocol specifics and the localisation of services in the protocol, the application developer is responsible solely for the development of the application logic.

In this chapter, an example application will be described and implemented to indicate the amount of effort required to create an application in the proposed architecture.

5.1 Join me in SecondLife!

As an application, consider the following scenario which is based on the example application described in section 1.3:

After a day of hard working, Alice decides to play some SecondLife. From all the activities that are going on in this virtual world, she chooses to visit a small club in ‘Virtual Holland’. After some time, she realises that she didn’t talk to Bob for quite some time and that he might be interested in joining her in this club. Since Bob does not show up in her ‘online friends’ list, she chooses to invite him to SecondLife through his personal buddy iCat, which is connected to the Internet. Alice can invite Bob via a small pop up within SecondLife.

After receiving the invitation, the ears of the iCat start glowing. Bob notices this and triggers the iCat to inform him about the received message. The iCat asks him whether or not Bob wants to join Alice online. Bobs answer is directly communicated to Alice, allowing here to redress before Bob enters the virtual world.

5.2 The setup

Neither the iCat nor SecondLife is designed explicitly to be used for these types of applications. The iCat is totally unaware of anything other than its controlling process, but this control process can be programmed in various programming languages, enabling the communication with other processes.

The iCat has a well defined interface [23]. The functionality that will be used in this application can be addressed as follows, where \([a]\) indicates that argument \(a\) is optional, and
\( a^* \) indicates that \( a \) may actually be a sequence of parameters \( a_1 \ a_2 \ \ldots \):

- **Load** channel animation [parameters]*
- **Start** channel repetitions
- **Set-var** variable value [frames]

The iCat makes use of animation files. An animation file defines the assignment of values to observable attributes (the position of the lips, color of the ears) as a function of the time. Loading an animation file in memory is done with the **Load** function. The channel identifier specified as a parameter of the load function is used later to **Start** the preloaded animation.

The observable attributes can also be assigned a value directly from the code. This is done with the **Set-var** function. The parameters consists of an attribute, the desired value and the duration of this assignment (e.g. assigning the color red to the iCats left ear for 3 frames looks as follows: set-var icat.head.leftear.light.red 255 3). In the current implementation, this value is chosen to have the default value \(-1\), which is equal to an infinite number of frames.

If one of the inputs of the iCat is triggered (the left or right paw is touched, in this example application), a notifying message is sent. In this application, the following sensor values are used:

- **leftpaw** touched
- **rightpaw** touched

The sensor values for these two sensors are Boolean values.

By itself, SecondLife can react to received XML-RPC messages, and it can initiate communication with the world outside SecondLife via HTTP requests. It is assumed that some basic functionality is made available via the use of these protocols. The following functions can be used:

- **Invite** Identifier
- **Response** Identifier comesOnline

The identifier parameters refer to the logical name assigned to devices in the system, which can be used to obtain the physical address of the device from the registry service 4.4.3. The **Invite** function sends an invitation to the device identified by **identifier**. The **Response** function is used to inform SecondLife about the decision made by the person addressed by an invitation.

### 5.3 Creating the application

The application connects the iCat with SecondLife. It handles the mapping between the inputs and outputs of the two components and ensures that the handling of input values is done conform the application scenario of Section 5.1. In figure 5.1, the connections that must be created by the application are shown as small boxes with a number in them.

The Tcl/Tk scripting language has been chosen to implement the application. This general purpose scripting language has been chosen for its capability to create non-terminating
5.3. CREATING THE APPLICATION

applications and the ease in which new components and functionality can be added during run time. A short introduction to the syntax of the scripting language can be found in appendix A.

The first thing that needs to be done when creating an application is to localise and connect the iCat and the SecondLife channel. From then on, the two services can be contacted and output values of these services are received. Establishing a connection is done in the following way, assuming that Alice identifies her avatar by ‘AliceAvatar’ and Bob has named his iCat ‘BobTheCat’:

set Alice [2ndLife AliceAvatar]
set Bob [iCat BobTheCat]

Setting the colors of the iCats ears and paws is an essential part of the user experience, but resetting every colour when one colour is (possibly) changed is cumbersome at least. To avoid this, a single function is defined that handles this for all LEDs:

proc SetBobColors {LE.R LE.G LE.B RE.R .. LP.R .. RP.R RP.G RP.B} {
    Bob Set-var icat.head.leftear.light.red $LE.R
    Bob Set-var icat.head.rightear.light.green $LE.G
    ...
    Bob Set-var icat.base.rightpaw.light.blue $RP.B
}

The function takes the RGB values (in the range 0..255) that are available for all user input sensors. The names of the input sensors are abbreviated and represent the LeftEar, RightEar, LeftPaw and RightPaw respectively.

The second step in creating an application is defining connections between services. According to the scenario, the interaction is initiated by Alice, by means of an invitation. As soon as that invitation is received, the iCat should indicate that there is a message waiting for Bob but wait with giving further explanation until user input is received.

# The connection that is labelled with a ‘1’ in the figure
when Alice Invite Bob {
    # Indicate that there is a message waiting

Figure 5.1: The connections between the iCat and SecondLife. The numbers represent pieces of code.
As soon as Bob presses the right paw of his iCat, the `AskBob` procedure is called. This procedure is not available in the framework and it will have to be defined:

```tcl
AskBob {question ifYes ifNo} {
    # Change the right paw to green and the left paw to red
    SetBobColors 0 0 0 ... 255 0 0 255 0
    Bob Set-var icat.speech.text $question

    # The connection that is labelled with a '2' in the figure
    when Bob leftpaw true [subst {
        $ifNo
            when Bob rightpaw true {}
        when Bob leftpaw true {}
    }]

    # The connection that is labelled with a '3' in the figure
    when Bob rightpaw true [subst {
        $ifYes
            when Bob rightpaw true {}
        when Bob leftpaw true {}
    }]
}
```

In Tk/Tk, functions can easily be passed as strings and created, deleted and modified during run time. This language feature is used repeatedly in the application logic described above. Functionality is only assigned to user inputs when this is necessary. If not, the event handlers are reset like it is done in the `AskBob` function.

The application in this chapter is not very complicated. It involves only two services that are combined in a fully sequential scenario. This has been done intentionally, such that the behaviour of the program can be grasped easily. However, the way in which the application logic is presented in this chapter is representative for the way in which the platform can be used to develop more interesting applications.
Chapter 6

Verifying data replication

In a real-world implementation of the registry described in section 4.4.3, a single machine might not be capable of handling all requests. Instead, several systems are placed in a cluster, named a *web farm*. In such a setup, the computational load can be balanced between systems, allowing for a more responsive system. A web farm allows to perform admission control and load balancing of transactions to available nodes [2, 33].

Load balancing is a powerful technique, distributing computation intensive tasks from a single point of entry to several systems. With load balancing, systems can handle more simultaneous requests than are possible when a single machine is used. To guarantee data consistency, all nodes can request the required data from a single machine, being the central data storage. Alternatively, the information can be shared among all connected machines and a data replication protocol can be used to keep the data consistent.

In this chapter, a protocol by Kemme and Alonso [22] regarding data consistency is presented. This protocol is modelled with mCRL2 and this model is used to verify several requirements on a data replication protocol.

6.1 Data consistency between distributed hosts

Data replication protocols orchestrate the handling of *transactions*, which are lists of *read* and *write* actions on *items*. The first categorisation of these type of protocols is into *lazy*, where replication is handled after a transaction is completed and *eager*, where replication is handled between the boundaries of a transaction. Although the lazy variant allows for faster response times, the eager variant leads to more consistent data [21, 4].

In a distributed database model, there is a collection of *n* nodes that keep a local record of data items and their values. Kemme and Alonso assume a static collection of nodes and assume that all data elements are available locally to every node.

Communication between nodes takes place via a group communication protocol, which handles communication and ordering. The messages are assumed to be totally ordered, meaning that all messages are delivered in exactly the same order at all nodes. A different ordering would be a causal one, where transactions are guaranteed to be delivered in the same order at all nodes if and only if their order is relevant to the consistency of the data. There are various approaches to implement totally ordered group communication protocols [22, 35].

Data consistency between nodes is guaranteed if all write operations, the only type of operation that influence consistency, are executed in the same order at all nodes. In that
sense, it would appear to an outside observer as if only one machine is maintaining the data. A protocol that behaves as if it consists of a single machine is called 1-copy-serializable.

6.2 A serializable protocol

Kemme and Alonso present three different protocols in their article [22], of which only one will be modelled here. The serializable approach has been taken in favour of cursor stability and snapshot isolation since it is the only one of the three protocols that prevents all possible isolation problems described by Berenson et al. [22, 3]. Be aware that the algorithm does not give any guarantees on the handling of transactions, due to the decision that every non-committed read action should be aborted whenever a write action on the same item is received. The other two protocols do not share this behaviour, but allow inconsistencies.

To avoid any ambiguity between the representation here and the protocol described by Kemme and Alonso, it is presented in figure 6.1 as it was listed in their article [22]. The algorithm works on items in the database and an arbitrary data item will be indicated by $X$.

The lock manager of each node $N$ controls and coordinates the operation requests of a transaction $T_i$ in the following manner:

1. **Local Reading Phase:** Acquire local read lock for each read operation $r_i(X)$. Defer write requests $w_i(X)$ until end of reading phase.

2. **Send Phase:** If $T_i$ is read-only, then commit. Else bundle all writes in $WS_i$ and multicast it (total order service).

3. **Lock Phase:** Upon delivery of $WS_i$, request all locks for $WS_i$ in an atomic step:
   a. For each operation $w_i(X)$ in $WS_i$:
      i. If there is no lock on $X$, grant the lock.
      ii. If there is a write lock on $X$ or all read locks on $X$ are from transactions that have already processed their lock phase, then enqueue the lock request.
      iii. If there is a granted read lock $r_j(X)$ and the write set of $WS_j$ of $T_j$ has not yet been delivered, abort $T_j$ and grant $w_i(X)$. If $WS_j$ has already been sent, then multicast abort message $a_j$ (basic service).
   b. If $T_i$ is a local transaction, multicast commit message $c_i$ (basic service).

4. **Write Phase:** Whenever a write lock is granted perform the corresponding operation.

5. **Termination Phase:**
   a. **Upon delivery of a commit message $c_i$:** Whenever all operations have been executed commit $T_i$ and release all locks.
   b. **Upon delivery of an abort message $a_i$:** Undo all operations already executed and release all locks.

Figure 6.1: Protocol for replication with serializability by Kemme and Alonso, taken from [22]

In order to model the protocol, the processes of which a node $N_i$ consists, as well as the actions that can occur should be defined. The separation into processes has been done by Kemme and Alonso [22]. Deciding which actions to include is done based on actions that are relevant to the requirements of the protocol. In the following section, the requirements will be introduced.
6.3 Requirements of the system

Kemme and Alonso list three properties that must hold for an eager replication protocol to be correct. The correctness of these properties is proven by applying structural induction on the shape of the protocol. Although this is sufficient, Kemme and Alonso do not define critical sections in their protocol, nor do they consider the behaviour of phases other than the lock phase in the analysis of their protocol in very much detail. By modelling the protocol in mCRL2, possible ambiguities are identified and remedied.

In this section, the desired properties formulated by Kemme and Alonso are reformulated as smaller and easier verifiable requirements on the model.

6.3.1 Absence of deadlock

One of the most desirable features in a protocol is the absence of deadlock. A model is said to contain a deadlock state if there is at least one configuration possible that is reachable from an initial state and from where no actions can be performed. A model is called *deadlock free* if no such state exists.

In a data replication protocol like the one considered here, no deadlock states should occur. For the original protocol description, Kemme and Alonso [22] proved that this is indeed the case. Although they did not put too much stress on the way in which the other phases should be implemented, the absence of deadlock should hold in the model of the protocol as well as in the original protocol.

Unfortunately, this does not seem to be the case for the model. In the protocol, transactions must be identifiable during the lifespan of the transaction. In order to make the system deadlock free, identifiers assigned to transactions should be released as soon as all nodes have completely finished the transaction. This way, the identifier can be reused and the number of used transaction identifiers can be limited, making the state space feasible.

In the current protocol, there is no place where it is known if all nodes have fully handled a specific transaction. Without this information, it is not possible to reuse a transaction identifier, since this may lead to duplicate transaction identifiers on nodes in the system.

Without modifying the protocol, the absence of deadlock can only be approximated by formalising several properties that make it reasonable that no deadlock states exist. The following requirements are added for that purpose:

**req 1.1** Every transaction identifier is assigned only once and to only one node.

**req 1.2** Every local transaction is eventually finished

**req 1.3** Every lock that is requested on any node is eventually released

6.3.2 Atomicity of transaction

When a node receives a transaction request, this request is called *local* to that node. Given that progress is made, at some point in time the transaction enters its lock phase at the local node and a decision is made whether the transaction should be committed or aborted.

Naturally, if consistency of data is of any interest, this decision should be followed by all other nodes and this is expressed in the following requirements:

**req 2.1** The decision to abort or commit a transaction is always made.
req 2.2 The decision to abort or commit a transaction is made by the node to which the transaction is local.

req 2.3 If a transaction has been committed by the local node, then all nodes commit this transaction.

req 2.4 If a transaction has been aborted by the local node, then all nodes abort this transaction.

req 2.5 The decision to abort or commit a transaction is not made by a node to which the transaction is not local.

6.3.3 1-Copy-serializability of the protocol

One of the main criteria for correctness of an eager replication protocol is the 1-copy-serializability [4]. This involves the system to behave as a single database forcing different concurrent transactions to be coordinated in such a way that their execution is equivalent with the sequential execution on a single copy of the data.

This involves only the write operations on the database, since the order in which read operations are executed does not influence the external behaviour of the database. In order to have 1-copy-serializability, all write operations should be operated on all nodes and should be operated in the same order.

More precisely, if the following requirements hold, then the protocol is 1-copy-serializable:

req 3.1 If transaction $t$ is going through its lock phase before transaction $t'$ on node $N$, then this order holds for all nodes.

req 3.2 All nodes request write locks for all transactions that are multicasted.

6.4 Model checking tools

To help in verifying required properties of the model, model checking algorithms can be used. A model checking algorithm takes a model $M = (\text{States}, \text{Init}, \text{Actions}, \text{Relations})$ and a formula $f$ describing a property as input parameters. The result of such an algorithm is a set of states $m, m \in M$ for which the property described in $f$ holds. To verify that a specific property is globally true, the number of states that is returned by the model checking algorithm should be equal to the number of states in the model.

There are several languages available to define the required properties of a model. Fairly well known is CTL*, with LTL and CTL as two subsets of this language. CTL*, as well as the subsets, works on Kripke structures, which define a set of states $S$, a set of initial states $I$, relations between nodes $R$ and a labelling function $L : S \rightarrow 2^{AP}$, giving the atomic propositions that hold in the supplied state. When not the atomic propositions of a particular state but the transitions between states are of interest, $\mu$-calculus can be used.

In the mCRL2 tool set, there are (experimental) tools available for verification. These tools use an extension to the $\mu$-calculus, allowing the inclusion of actions with parameters in the formula (First Order Modal $\mu$-calculus). When the term $\mu$-calculus is used in the remainder of this chapter, this extension is meant instead. The formal syntax is currently
only documented in an internal document [28] due to the currently experimental nature of the tool set. Some examples of the syntax are given by Groote and Willemse [14].

6.4.1 An introduction to $\mu$-calculus

The $\mu$-calculus is a mathematical language that can be used to define propositions on states and transactions between states in a model. The model $M = (S, T, L)$ on which $\mu$-calculus formulae are evaluated consists of all states $S$ of the model, the transitions between states $T$ and a mapping of atomic propositions that are true in a particular state $L$.

In model checking, the validity of a property $p$ in a state $s$ may depend on the validity of property $p'$ ($p$ and $p'$ may be equivalent) in state $s'$. When state $s$ validates $p$, changes in the collection of states where property $p'$ holds influences the validity of $p$ in $s$. As a remedy, states and properties are re-evaluated until no further changes occur. The resulting collection of states is called a fixpoint and this concept is frequently used in the formalisation of the requirements.

In $\mu$-calculus, there are two different types of fixpoints. The least fixpoint operator $\mu$ returns the fixpoint that is the least among all fixpoints. The greatest fixpoint operator $\nu$ returns the fixpoint that is the maximum among all fixpoints. Finding the fixpoint of a $\mu$-calculus formula in a model is done by giving an approximation of the states in which the process holds, and refine this approximation by repeating the process. In a finite model, this process is guaranteed to end. Additionally, the provided formula should be monotonic in its behaviour in order to be semantically meaningful [10].

Two other operators that are available, apart from the propositional logic operators like conjunction, disjunction and negation, are the $[a]$ and $\langle a \rangle$, with $a$ an action. They are used to express conditions on transitions and the states to which they lead. Given that $\llbracket f \rrbracket_M e$ represents the interpretation of $f$ in the model $M$: 

$$\llbracket \langle a \rangle f \rrbracket_M e = \left\{ s | \exists t \left[ s \xrightarrow{a} t \text{ and } t \in \llbracket f \rrbracket_M e \right] \right\}$$

$$\llbracket [a] f \rrbracket_M e = \left\{ s | \forall t \left[ s \xrightarrow{a} t \text{ and } t \in \llbracket f \rrbracket_M e \right] \right\}$$

As an example calculation of a $\mu$-calculus formula, the following requirement will be rewritten and verified over the simple model, presented in figure 6.2.
1. All paths inevitably include an $a$-labelled transition.

Rewriting this requirement to an equivalent $\mu$-calculus formula results in the following:

$$\mu X. \langle \top \rangle \top \land \neg a \land X$$

If this formula is approximated once for every state present in the model of figure 6.2, only a single state will match the formula. During the second approximation, using the states found in the first approximation as the set of states $X$, the initial state is added to the set of resulting states as well. Further evaluations do not add any states to this set, making the two states currently found the fixpoint of the formula. Since the initial state is element of the set of states that are the result of evaluating the formula, it can be stated that, from the initial state, all paths must include an transition labelled with an $a$.

### 6.4.2 The mCRL2 tool set

The mCRL2 tool set is still under active development and is expected to contain some defects. As such, the exact configuration as it was used during modelling and verification is presented here. This can safely be skipped for readers not familiar with this tool or uninterested in the version numbers and is included for reference purposes only.

The tools used are listed in table 6.1. The version number that is given was retrieved by passing the ‘–version’ parameter to the executable.

<table>
<thead>
<tr>
<th>tool</th>
<th>arguments</th>
<th>version</th>
</tr>
</thead>
<tbody>
<tr>
<td>mcrl2lps</td>
<td></td>
<td>revision 3599-release-shared</td>
</tr>
<tr>
<td>lps2lts</td>
<td></td>
<td>revision 3599-release-shared</td>
</tr>
<tr>
<td>ltsconvert</td>
<td>-u</td>
<td>revision 3599-release-shared</td>
</tr>
<tr>
<td>ltsgraph</td>
<td></td>
<td>revision 3599-release-shared</td>
</tr>
<tr>
<td>lps2pbes</td>
<td>-f</td>
<td>revision 3599-release-shared</td>
</tr>
<tr>
<td>pbes2bool</td>
<td></td>
<td>revision 3599-release-shared</td>
</tr>
</tbody>
</table>

Table 6.1: The context in which the verification has taken place.

The language mCRL2 is a specification language used to specify and analyse the behaviour of distributed systems and protocols [9]. A model written in this specification language can be converted into an equivalent linear process with the use of mcrl2lps. This linear process contains no parallelism and is the basis input for most tools in the tool set. In the analysis of the model in this chapter, two tools that use a linear process as an input have been used. For the model checking, the linear process together with a formula in $\mu$-calculus can be used to generate a Parameterised Boolean Equation Systems (PBES) with the lps2pbes tool. The validity of the formula in the model can be obtained by solving the PBES, resulting in a Boolean value, with the pbes2bool tool. The linear process can also be used to generate the state space of the model with the lps2lts tool. The resulting state space can be rewritten in another format, possibly after applying a minimising method such as branching bisimulation, with the ltsconvert tool. Visualising the state space in a two-dimensional graph can be done with the ltsgraph tool.
6.5 The actions of the model

To verify the requirements stated in section 6.3, these requirements must be formalised using the \( \mu \)-calculus. In \( \mu \)-calculus, the exact names of transitions, and their parameter values, play an important role. Consequently, in order to formalise the requirements, the actions that are available in the model must be known. In this section, the available actions will be described.

6.5.1 The observable actions

The communication with external processes is shown in figure 6.3, where arrows have been drawn wherever Kemme and Alonso indicate communication with processes outside the node.

![Figure 6.3: The communication of a node with external processes](image)

There are three different processes with whom the node is communicating. The exact actions and their parameters will be described in the following subsections.

**Communication between node and user**

Users are processes that are interested in the information that is available in the database. To retrieve that information, the user process requests the information from a single node that is part of the system. Any request from a user process, named a *transaction*, consists of a list of read and write operations on one or more items in the database.

In communication with the node, a user process can send a request and receives a result when the request is handled. These two actions are considered to be the only observable actions of the user process, having the following names and parameters:

<table>
<thead>
<tr>
<th>Action name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UQ((n:\text{Node}, t:\text{Transaction}))</td>
<td>The action UQ (<em>User Query</em>) contains a transaction request ( t ). The type Transaction represents an arbitrary list of read and write operations on items in the database.</td>
</tr>
<tr>
<td>UR((n:\text{Node}, t:\text{Transaction}, s:\text{Status}))</td>
<td>The action UR (<em>User Result</em>) communicates with the user the result of evaluation of transaction ( t ). The status parameter ( s ) is a value indicating whether or not the transaction was committed or aborted.</td>
</tr>
</tbody>
</table>
In these descriptions, currently undefined types are used. These types will be defined after the requirements have been formalised. In this way, the number of values that belong to a specific type can be minimised such that the size of the state space that is generated from the model can be kept reasonably small.

Communication between node and network

The group communication medium shown in figure 6.3 is assumed to offer reliable and in-order delivery of messages to all nodes (including the sending node) connected to the medium. Following the protocol description in figure 6.1, two different types of messages, transactions and the decision to abort or commit by the local node, can be sent to all other nodes via the group communication medium. In the model, these communication actions are represented by the following actions:

<table>
<thead>
<tr>
<th>Action name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send(n:Node, t:Transaction)</td>
<td>Sends transaction $t$ to all nodes connected to the communication medium. All communication initiated by this operation is considered to be reliable and totally ordered.</td>
</tr>
<tr>
<td>Send(n:Node, t:Transaction, r:Result)</td>
<td>Sends the decision to abort or commit transaction $t$ to all nodes connected to the communication medium. Communication initiated by this operation is considered to be reliable but does not have to arrive in-order.</td>
</tr>
<tr>
<td>Receive(n:Node, t:Transaction)</td>
<td>Indicates transaction $t$ to be received from the group communication medium by the node identified by $n$.</td>
</tr>
<tr>
<td>Receive(n:Node, t:Transaction, r:Result)</td>
<td>Indicates that node $n$ receives the decision $r$ to abort or commit transaction $t$.</td>
</tr>
</tbody>
</table>

Communicating with the database

The communication with the database is handled by the Lock Manager. As such, no direct communication with the database is required and it is therefore omitted from figure 6.3. The internal communication with the lock manager will be introduced next.

6.5.2 The internal actions

Apart from the behaviour that is observable from the outside of the node, there is also some interesting functionality hidden inside the node. The communication, transaction and lock manager process of the node each run in their own thread but can communicate information and requests whenever this is required. The interfaces of the three internal processes shown in figure 6.4 will be described here.
6.5. THE ACTIONS OF THE MODEL

Communicating with the Lock Manager

The database contains all items that should be kept consistent among all nodes in the system. Communication with the database is done via the Lock Manager, that can request or release locks on items in the database. A lock request can either be granted or rejected, depending on the current locks on items (see figure 6.1, lock phase).

In the model, the operations regarding lock requests, lock responses and lock releases are modelled as the following actions:

<table>
<thead>
<tr>
<th>Action name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock(n:Node, t:Transaction, a:Action)</td>
<td>Request the locks of transaction t on node n. The action a indicates the type of the locks and can be equal to 'read' or 'write'.</td>
</tr>
<tr>
<td>LockResult(n:Node, t:Transaction, r:Result)</td>
<td>Returns the result of a lock request. A lock request for transaction t on node n can be either granted or rejected.</td>
</tr>
<tr>
<td>LockDecision(n:Node, t:Transaction, r:Result)</td>
<td>Requests the lock manager of node n to abort or commit transaction t, depending on r. The committing or abortion of a transaction is not handled immediately, but only after the write set of the message has been received.</td>
</tr>
<tr>
<td>ClearLock(n:Node, t:Transaction)</td>
<td>Requests the lock manager of node n to remove all read and write locks on transaction t, without making any changes on the data in the database.</td>
</tr>
</tbody>
</table>

There are three cases distinguished in the protocol when requesting a lock for a particular item. If there is no lock present, then the lock is granted. If one or more locks exist, but these locks belong to transactions that have already passed their lock phase, then the lock request is added to the queue of lock requests for this particular item. As a final option, a...
lock request for a transaction that has not yet passed its lock phase can be present. This lock request must belong to a local transaction (it can only be a read lock, which are only observable locally) and as such, this transaction must be aborted.

The three different decisions are represented by the following actions in the model:

- **Action name**: NoLocks($n$:Node, $t$:Transaction)
  - **Description**: Internal operation, indicates that there are no locks that conflict with the current lock request on node $n$.

- **Action name**: ConflictingLocks($n$:Node, $t$:Transaction)
  - **Description**: Internal operation, indicates that there is a lock request that conflicts with transaction $t$ on node $n$.

- **Action name**: NoConflictingLocks($n$:Node, $t$:Transaction)
  - **Description**: Internal operation, indicates that there is no lock request that conflicts with transaction $t$ on node $n$.

As soon as the lock manager has received the write set and the decision message of a particular action, it can handle the transaction and release the locks. This is done with the following internal action:

- **Action name**: removeLocks($n$:Node, $t$:Transaction)
  - **Description**: Internal operation, executed when both the decision and the lock request of a particular transaction $t$ have been received on node $n$.

**Communication with the communication manager**

The communication manager is a process that receives messages from the network. As soon as a write set or result message is received, it is handled as is described in the protocol of figure 6.1 (the lock phase and termination phase).

Most of the interface of the communication manager is symmetrical to that of the lock manager. There is one action that allows the communication manager to inform the transaction manager of the decision made by the lock manager on a write set of a particular transaction.

- **Action name**: LockResultWS($n$:Node, $t$:Transaction, $r$:Result)
  - **Description**: Communicates the result $r$ of requesting the locks for the write set of transaction $t$ to the transaction manager of node $n$.

**Communication with the transaction manager**

In the transaction manager, the operations that are assigned to the different phases of the protocol presented in figure 6.1 are executed. It is the only place where the distinction between local and remote transactions is made and messages to all other nodes are sent from here.
Any conflicting lock is communicated to the transaction manager. The transaction manager can send an abort message to the group communication medium if required. The action allowing to inform the transaction manager is modelled as follows:

| Action name: | RejectLock(n:Node, t:Transaction) |
| Description: | Tells node n that transaction t, which is a local transaction that has not yet passed its lock phase, must be aborted. |

### 6.6 Formalising the requirements

With the observable actions defined, the requirements can be formalised into μ-calculus. The formalisation of requirements will be done in this section, followed by the model of the protocol and the evaluation of the requirements in the following sections.

The requirement identifiers used in this section refer to the identifiers used in section 6.3, all actions used are presented in section 6.5. Whenever the relation between a requirement and its representation in μ-calculus is not directly clear, the reasoning to get to this formula is given as well.

#### 6.6.1 Abstractions of the model

In order to keep the state space generated by the model feasible, some abstractions will have to be made. Since the abstractions might be usable as assumptions in the verification of the requirements, they are presented here before the actual formalisation of the requirements.

A transaction is a list, containing any number and order of read and/or write operations. As a result, the Transaction data type consists of an infinite number of data elements. Using them when generating a state space results in a state space explosion from the initial state on.

Fortunately, the exact operations that are part of an transaction are of little importance to the way in which the protocol operates. It is required that transactions have an identifier and that this identifier is unique. An enumerable type is sufficient, since the number of transactions that will be initiated by the user process is bounded. It is assumed that in the initial state a random distribution of locks is present. This way, the general behaviour of the protocol can be observed with only a bounded number of transactions initiated by users of the system.

#### 6.6.2 Absence of deadlock

**req 1.1** Every transaction is uniquely identified, so the requirement holds if no assignment of a transaction identifier t to a node, represented in the model by the action \(UQ(t)\) is followed by the assignment to the same transaction to any, possible other, node.

In μ-calculus, the requirement is formalised as follows:

\[ \nu X. \left[ \top \right] X \land \forall n:\text{Node} \forall t:\text{Transaction} \left[ UQ(n, t) \right] \left( \nu Y. \left[ \top \right] Y \land \exists n':\text{Node} UQ(n', t) \right) \bot \]
CHAPTER 6. VERIFYING DATA REPLICATION

req 1.2 A transaction $t$ is finished if the action that introduced the transition in the system, $UQ(n, t)$, is inevitably followed by a corresponding action $UR(n', t)$, for some $n, n' \in Node$. A transition is inevitable from a particular state $s$ if all paths starting from $s$ contain a state $s'$ from which the only available transition is the action $UR(m)$. Modelling this in $\mu$-calculus results in the following formula:

$$\nu X. [T] X \land \forall t: Transaction \exists n: Node UQ(n, t) \langle \mu Y. \top \land \neg \exists n': Node \exists p: Phase UR(n', t, p) \rangle Y$$

req 1.3 All locks are released if, for any node, every call to $Lock(node, transaction, a)$, for $a \in \{read, write\}$ is inevitably followed by a call to $ClearLock(node, transaction)$ or $removeLocks(node, transaction)$.

Modelling this in $\mu$-calculus results in the following formula:

$$\nu X. [T] X \land \forall n: Node \forall t: Transaction \forall a: Action [Lock(n, t, a)] (\mu Y. \top \land \neg \exists n: Node \exists p: Phase UR(n, t, p)) Y$$

6.6.3 Atomicity of transactions

req 2.1 All transactions that are offered to the system will eventually be committed or aborted. Note that this requirement is a less restrictive case of req 2.2. If the validity of req 2.2 can be shown, the validity of req 2.1 is proven automatically.

req 2.2 The decision to abort or commit a transaction should be always made and is made by the node to which the transaction is local. The result of this decision is communicated to the user by the $UR(node, transaction, decision)$ action. If this requirement is formalised to the $\mu$-calculus, the following formula is obtained:

$$\nu X. [T] X \land \forall n: Node \forall t: Transaction UQ(n, t) \langle \mu Y. \top \land \neg \exists p: Phase UR(n, t, p) \rangle Y$$

Apparently, this is roughly same formula as was obtained for req 1.2, additionally requiring that the transaction response is send by the same node that initially received the transaction request. It might be interesting to further restrict the requirement to the case where the decision is communicated to the user process exactly once, and that no other process communicates this decision to the user process.

$$\nu X. [T] X \land \forall n: Node \forall t: Transaction UQ(n, t) \langle \mu Y. \top \land \neg \exists p: Phase UR(n, t, p) \rangle Y \land \forall n': Node. (n \neq n') \Rightarrow [\exists p: Phase UR(n', t, p)] \bot$$

This formula expresses that every action $UQ(n, t)$ is inevitably followed by exactly one action $UR(n, t, p)$ and that there is no action $UR(n', t, p)$, for $n' \neq n$. 
6.6. FORMALISING THE REQUIREMENTS

req 2.3 and req 2.4  The effect of a transaction is supposed to be made permanent after the transaction locks have been released. This holds both for the aborted and the committed transactions. The decision of a local node can be made silently, if the write set of that transaction has not yet been sent to the group communication medium. Otherwise, the effect of a transaction must be made permanent on all nodes. This is formalised as:

$$\forall n \in \text{Node} \left( \forall X. \left[ \top \right] X \land \forall n'. \forall t. \forall r. \exists \text{Result} \left[ \text{Send} \left( n', t, r \right) \right] \left( \forall Y. \left[ \top \right] Y \land \neg \text{removeLocks} \left( n, t, r \right) \right) \right)$$

This formula expresses that every action \( \text{Send} \left( n', t, r \right) \) is eventually followed by a \( \text{removeLocks} \left( n, t, r \right) \) action, for every \( n \in \text{Node} \).

req 2.5  This requirement holds if no node except the node to which the transaction is local executes the action \( \text{Send} \left( \text{node}, \text{transaction}, \text{result} \right) \)

$$\forall X. \left[ \top \right] X \land \forall n. \forall t. \exists \text{Result} \left[ \text{Send} \left( n', t, r \right) \right] \left( \forall Y. \left[ \top \right] Y \land \forall n'. \forall t. \forall r. \left( n \neq n' \Rightarrow \exists \text{Result} \left[ \text{Send} \left( n', t, r \right) \right] \right) \right)$$

6.6.4  1-copy-serializability

req 3.1  Formulating this requirement in the \( \mu \)-calculus is more complicated than suggested by the possible number of sequences. In verifying this requirement, only the relative order of the \( \text{Lock} \left( n, t, \text{write} \right) \) transition label instances, for \( n \in \text{Node}, t \in \text{Transaction} \), are interesting.

The order between instances of this transition label can be obtained by hiding all actions other than the \( \text{Lock} \left( n, t, \text{write} \right) \) action and calculating the weak trace equivalent graph of the original model. In a weak trace equivalent graph, all internal (hidden) actions that remain after branching bisimulation are simply removed, effectively minimising the graph to the actions that are observable.

This results in the graph shown in figure 6.5. Note that a currently unknown transition label, namely \( \text{cLockW} \) is shown. This label represents the simultaneous execution (communication) of the \( \text{Lock} \left( n, t, a \right) \) action by two processes, with \( a = \text{write} \). This modification was required to distinguish between the \text{read} and \text{write} parameter values and does not influence the behaviour of the model.

It follows from the graph in figure 6.5 that write locks are requested in the same order on every node, which validates the requirement.

Alternatively, a parameterised \( \mu \)-calculus formula has been suggested by Tim Willemse. According to his experiments, this formula expressed the requirement and was evaluated as being valid in the model. Because the tools used in the verification used for the other requirements were incapable of reproducing this result, the formula has been included in appendix B but will not be presented here.

req 3.2  For this requirement to hold, all possible paths must include the action \( \text{Lock} \left( n, t, \text{write} \right) \) for all \( n \in \text{Node} \) and every \( t \in \text{Transaction} \) that has been multicasted via \( \text{Send} \left( \text{node}, t \right) \). Formalising this in \( \mu \)-calculus results in the following:
Knowing the actions that are present in the protocol, together with their parameters, it is now possible to model the protocol using the mCRL2 modelling language. In this section, the node process is modelled. The node process consists of three smaller processes as was shown in figure 6.4.

Within the processes, a distinction is made between sending and receiving an action, or the communication of that action if both are performed simultaneously. This distinction is made by prefixing the action with the characters $s$, $r$ and $c$.

In the verification of the protocol, not all possible configurations can be considered. Therefore, a configuration is selected that is sufficiently large to be representative yet small enough to result in a feasible state space. In this configuration, the system consists of two nodes and is given two transactions.

Considering two nodes is sufficient in modelling the behaviour of the protocol, since the only distinction between transactions is made in terms of ‘local’ versus ‘remote’. Two transac-

\[
\forall_{n:Node} (\nu X. [\top] X \land \forall_{n':Node} \forall_{t:Transaction} [\text{Send}(n', t)] (\mu Y. (\top) \top \land [\neg \text{Lock}(n, t, \text{write})] Y ))
\]

6.7 Modelling the protocol

Figure 6.5: A weak trace equivalent version of the model in which all actions other than $c\text{Lock}W$ are hidden.
6.7. MODELLING THE PROTOCOL

tions is the minimal number of transactions required to ensure that a conflicting lock requests may occur, which is essential in the data replication protocol.

6.7.1 Types used in the model

In the description of the actions, several types that are not normally available in the mCRL2 language are used. Before the different processes will be modelled, the more interesting types are introduced.

Every node consists of three different processes that communicate. The different processes must be able to identify the other processes that belong to the same node. Also, the group communication protocol uses this information to address the different nodes.

Node = struct N1 | N2;

As observed in section 6.6.1, transactions need to be identified by a unique identifier. As explained earlier, there are only two transactions required so this is easily modelled.

Phase = struct read | send | lock | committed | aborted;
Transaction = struct t1 | t2;
TransactionItem = struct tr(identifier:Transaction, phase:Phase);

The delivery of messages by the group communication medium requires more information than there is present in the TransactionItem type. The node to which to send the message is required to guarantee that a message is sent at most once. This type is defined as follows:

TransactionQueue = struct tq(node:Node, transaction:TransactionItem);

6.7.2 Functions

Processes in the model may manipulate the values that are passed as parameters. For this purpose, several functions have been defined. Their implementation in mCRL2 can be found in appendix B.

The filter operation takes a list \( L \) and a desired value \( v \) for property \( p \) as its arguments and returns a list of matching elements. With \( \triangleright \) representing list concatenation and \( [a_1, a_2, \ldots] \) representing a list with elements \( a_1, a_2, \ldots \), the filter function is defined recursively as follows:

\[
\text{filter}(L, v) = \begin{cases} 
\text{filter}(\text{tail}(L), v) & \text{if } p(\text{head}(L)) \neq v \\
[\text{head}(L)] \triangleright \text{filter}(\text{tail}(L), v) & \text{if } p(\text{head}(L)) = v
\end{cases}
\]

The remove operation takes a list \( L \) and a single element \( t \) of the same type as the elements in the list. This is defined recursively as follows:

\[
\text{remove}(L, t) = \begin{cases} 
[\text{head}(L)] \triangleright \text{remove}(\text{tail}(L), t) & \text{if } \text{head}(L) \neq t \\
\text{remove}(\text{tail}(L), t) & \text{if } \text{head}(L) = t
\end{cases}
\]

The third operation, update, takes a list \( L \), a single element \( t \) of the same type as the elements of \( L \) and a value \( v \) for known property \( p \). The \( \text{update}(L,t,v) \) operation is defined recursively as follows:

\[
\text{update}(L, t, v) = \begin{cases} 
[\text{head}(L)] \triangleright \text{update}(\text{tail}(L), t, v) & \text{if } \text{head}(L) \neq t \\
[\text{head}(L)(p := v)] \triangleright \text{update}(\text{tail}(L), t, v) & \text{if } \text{head}(L) = t
\end{cases}
\]
6.7.3 A single node

A node is the parallel composition of three processes, as shown in figure 6.4. In this section, these three processes will be defined.

The transaction manager

The transaction manager keeps score of the states of the transactions that are local to this node.

When a new transaction is received from the user process, it is added to the list of local transactions. According to the protocol, the transaction is waiting to enter its read phase.

```
proc NODE_TRANSACTION_MANAGER(n:Node, local:List(TransactionItem)) =
  sum t:Transaction .
  rUQ(n,t) .
  NODE_TRANSACTION_MANAGER(n, local ++ [tr(t,read)])
```

As soon as a transaction has entered its read phase, the read locks for the transaction are requested. Read locks are always granted eventually and the transaction is marked to be ready to enter its send phase.

```
+ sum t:Transaction .
  (tr(t,read) in local)
  -> sLock(n,t,r) .
  sum r:Result .
  rLockResult(n, t, r) .
  NODE_TRANSACTION_MANAGER(n, update(local, tr(t,read), send))
<> delta
```

When the transaction manager allows a transaction to enter its send phase, the transaction is communicated to the group communication medium. The transaction has successfully requested read locks and is waiting for the Write Set message to be delivered.

```
+ sum t:Transaction .
  (tr(t,send) in local)
  -> sSend(n,t) .
  NODE_TRANSACTION_MANAGER(n, update(local, tr(t,send), lock))
<> delta
```

The communication manager is running in parallel with the transaction manager. While handling a local transaction, write sets of remote transactions can be received. When this occurs, write locks for this write set are requested by the communication manager.

These write locks may conflict with read locks of local transactions. If this is the case, the local transaction is aborted by the transaction manager. Two cases are distinguished that influence the way in which the conflict is handled. In the first case, the write set of the transaction has not yet been sent to the group communication medium. The user that initiated this transaction can be informed about the abortion of the transaction and it can be removed from the list of local transactions.
6.7. MODELLING THE PROTOCOL

In the second case, the write set of the transaction has already been communicated to the group communication medium. The local node decides that the transaction causes conflicts and sends an abortion message to the group communication medium.

This behaviour is modelled in mCRL2 as follows:

\[\text{sum } t: \text{Transaction} .\]
\[\text{rRejectLock}(n,t) . (\]
\[\text{tr}(t, \text{send}) \text{ in local} \]
\[\rightarrow \text{sLockDecision}(n,t, \text{abort}) .\]
\[\text{sClearLock}(n,t) .\]
\[\text{SUR}(n,t, \text{aborted}) .\]
\[\text{NODE\_TRANSACTION\_MANAGER}(n, \text{remove}(\text{local}, \text{tr}(t, \text{read})))\]
\[\text{<> delta}\]
\[+\]
\[\text{tr}(t, \text{lock}) \text{ in local}\]
\[\rightarrow \text{sSend}(n,t, \text{abort}) .\]
\[\text{SUR}(n,t, \text{aborted}) .\]
\[\text{NODE\_TRANSACTION\_MANAGER}(n, \text{remove}(\text{local}, \text{tr}(t, \text{lock})))\]
\[\text{<> delta}\]
\]

The communication manager handles the write sets of remote and local transactions. The result of the lock request of a write set influences the node to which the transaction is local in committing or aborting the transaction. The communication manager does not know which transactions are local and communicates all lock request results to the transaction manager.

\[+\text{sum } t: \text{Transaction} .\]
\[\text{sum } r: \text{Result} .\]
\[\text{rLockResultWS}(n,t,r) . (\]
\[\text{tr}(t, \text{lock}) \text{ in local}\]
\[\rightarrow ((r == \text{abort})\]
\[\rightarrow \text{sUR}(n,t, \text{aborted})\]
\[\rightarrow \text{sUR}(n,t, \text{committed})\]
\[\text{).}\]
\[\text{sSend}(n,t,r) .\]
\[\text{NODE\_TRANSACTION\_MANAGER}(n, \text{remove}(\text{local}, \text{tr}(t, \text{lock})))\]
\[\text{<> NODE\_TRANSACTION\_MANAGER}(n, \text{local})\]

The communication manager

The communication manager is waiting for the group communication medium to send a message. As soon as a write set of a transaction is received, the appropriate locks are requested from the lock manager. The result of this lock request is communicated with the transaction manager, as explained in the previous section.

\[\text{proc } \text{NODE\_COMMUNICATION\_MANAGER}(n: \text{Node}) =\]
\[\text{sum } t: \text{Transaction} .\]
\[\text{rReceive}(n, t) .\]
sLock(n, t, w) .
sum r:Result .
  rLockResult(n, t, r) .
  sLockResultWS(n,t,r) .
  NODE_COMMUNICATION_MANAGER(n)

If the communication manager receives the result of a transaction, it releases the locks that belong to that transaction.

sum t:Transaction .
sum r:Result .
  rReceive(n, t, r) .
  sLockDecision(n,t,r) .
  NODE_COMMUNICATION_MANAGER(n);

The lock manager

The lock manager receives lock requests from the other two processes inside the node process. Lock requests for a read operation are always granted, but lock requests for write operations can lead to conflicts with locks that are already present. In case of conflicting locks, the conflicting lock must be a read lock for a local transaction (see the protocol). This local transaction should be aborted via the transaction manager.

In the modelling of the lock manager, any arbitrary configuration of items and locks is assumed. A consequence of this is that no information about current locks needs to be stored, since it is unknown what the effect will be on the current configuration. However, when a conflicting lock is found, a previous transaction must be aborted. In the model, transactions are stored in any of three parameters. The readLocks parameter is a list containing all transactions that potentially have read locks present. As soon as a lock request for the write set of a transaction is received, the transaction is removed from the list of readLocks (the read locks are still present, but cannot lead to conflicting situations any more) and added to the list of write locks.

proc NODE_LOCK_MANAGER(n:Node, readLocks:List(Transaction),
                        writeLocks:List(Transaction), releases:List(Transaction)) =

  sum t:Transaction . ( sum a:Action .
                        rLock(n,t,a) . (a == r) -> sLockResult(n,t,commit) . NODE_LOCK_MANAGER(n,
                                                                   readLocks ++ [t], writeLocks,releases)
                        <> ( NoLocks(n,t) .
                             sLockResult(n,t,commit) .
                             NODE_LOCK_MANAGER(n, remove(readLocks,t),
                                               remove(writeLocks,t) ++ [t], releases)
                           +
                             NoConflictingLocks(n,t) .
                           ) )

sLockResult(n,t,commit).
NODE_LOCK_MANAGER(n, remove(readLocks, t),
remove(writeLocks,t) ++ [t], releases)
+
(#remove(readLocks,t) != 0)
-> ConflictingLocks(n,t).
  sRejectLock(n, head(remove(readLocks, t))).
  sLockResult(n,t,commit).
NODE_LOCK_MANAGER(n, remove(readLocks, t),
remove(writeLocks,t) ++ [t], releases)
<> delta
)
)

Apart from lock requests, the lock manager can also receive decisions about the abortion or committing of transactions. These decision messages must be stored until the write set of the corresponding transaction has been received.

When the transaction manager decides that a transaction should be aborted and the write set of this transaction has not been sent to the group communication medium, its locks should be removed. The ClearLock action handles this as follows:

+ sum r:Result .
  rLockDecision(n,t,r).
  NODE_LOCK_MANAGER(n, readLocks, writeLocks, releases ++ [t])
+
rClearLock(n,t). NODE_LOCK_MANAGER(n,
remove(readLocks,t),
remove(writeLocks,t),
remove(releases,t)
)

Eventually, both the write set and the decision of a single transaction have been received. At this point in time, the decision can be applied to the local copy of the data and the locks can be removed.

+ (t in writeLocks && t in releases) ->
  removeLocks(n,t). NODE_LOCK_MANAGER(n,
  remove(readLocks,t),
  remove(writeLocks,t),
  remove(releases,t)
  )
<> delta
);
6.7.4 External processes

The data replication protocol must work for any number of nodes, so in particular for a single node. In practice, using a data replication protocol is only interesting if multiple nodes are present in the system. If the parallel execution of two node processes is evaluated, modifications of the protocol are required to solve problems with respect to giving unique identifiers to transactions and providing total order group communication. To avoid these modifications of the protocol, a user process and group communication process are modelled, providing just enough functionality as is desired by the data replication protocol. In this section, the two additional functionality processes will be described.

The user

The user process represents a process that initiates transactions by sending a transaction query to any node. The process takes a list of available transaction identifiers and uses every identifier exactly once.

In their protocol description, Kemme and Alonso did not limit the number of user processes to strictly one, nor did they define the way in which process identifiers should be used. In the model, the process description as presented here is convenient because it guarantees that every identifier is unique. Rewriting the model such that multiple user processes are involved is straightforward and will not be further investigated in this chapter.

\[
\text{proc USER}(t:\text{List(Transaction)}) =
\]
\[
(\text{#}t > 0)
\]
\[
\rightarrow \sum n:\text{Node} . \ sUQ(n, \ \text{head}(t)) . \ \text{USER}(\text{tail}(t))
\]
\[
\leftrightarrow \delta;
\]

Group communication

The group communication medium provides the functionality to broadcast a message to all nodes connected to the group communication medium. Additionally, all messages containing a transaction should be delivered in the same order at every connected node, where the order of the messages containing the decision to abort or commit a transaction can be delivered in any order. The group communication process can be split in two parts, one for receiving messages from the nodes and one part to send the messages to all nodes:

The receiving part receives either Send(node, transaction) or Send(node, transaction, result) messages from the nodes. In one atomic action, messages received are added to the end of the queue for every node in the system. This guarantees that the ordering of transactions and results in the queue is the same for all nodes.

\[
\text{proc NETWORK}(\text{transactions: List(TransactionQueue)}, \\text{results: List(TransactionQueue)}) =
\]
\[
\sum n:\text{Node} . (\ \\sum t:\text{Transaction} . (\ rSend(n,t) . \ \text{NETWORK} (\ \text{transactions ++ [tq(N1,\text{tr}(t,lock)), tq(N2,\text{tr}(t,lock))]}, \ \text{results}
\]

6.8. Evaluation of the requirements

In the evaluation, a system with two nodes, handling two transactions is assumed. The state space generated by the parallel composition of two node processes, a network process and a user process consists of 46876 states and 140722 transitions. There is only a single state in which transactions are delivered should be the same on all nodes. To guarantee this, a filter is applied over all messages in the queue, returning the messages that are addressed to a particular node in the same order as in the original queue. From the queue that is the result of this filter, the head element is taken and the identifier of the transaction that is on top of the queue is send to the node.

The result messages that are broadcasted via the group communication protocol may be delivered in any order, which is expressed by the following mCRL2-code.

All communication in mCRL2 is considered to be reliable, implying that messages that have been communicated (i.e. the corresponding action has been performed) can be safely removed from the list of transactions.
which deadlock holds, which is where the user process can no longer execute the $UQ$ action. All other processes are in the same state as they were initially.

The formalisation of the requirements in the $\mu$-calculus has been used in the generation of Parameterised Boolean Equation Systems (PBES), that have been evaluated. Following the results of this evaluation, all requirements are considered to be valid. This indicates that the model as it has been introduced in this chapter gives a correct implementation of the replication protocol suggested by Kemme and Alonso.

The model described here only matches the protocol and does not fully match the implementation used by Kemme and Alonso. Throughout their paper, several decisions are made on the way a transaction is applied to the items in the data storage. The model generalises over these decisions by combining them in a single action and it is possible that an action actually represents several choices. Other differences may relate to the implementation of the side processes. In their protocol, Kemme and Alonso state that users can autonomously decide which node receives their requests. In the model, a single user process randomly distributes transaction requests in order to uniquely identify a transaction. Different approaches are possible, but considered equivalent during the modelling. The group communication protocol is more interesting. Kemme and Alonso frequently use the total order semantics of the group communication medium in their proves and arguments, but do not specify the protocol that was used for this group communication medium.

Concluding, the desirable properties were proven to be valid for the presented model. The experimental nature of the tools that were used, together with the verbose description of the protocol, spread over several pages and articles, makes it difficult to state that the same holds for the original protocol. This does not decrease the value of the verification done in this chapter, as it gives a concise description of the processes involved which can be used in actual implementations of the protocol, but the results of the verification would be more interesting if the protocol and the model would be guaranteed to be identical.
Chapter 7

Conclusion

7.1 The proposed solution

The platform proposed in this thesis is a standard layered solution to a distributed application. There are three layers, each with its own functionality. The communication layer handles communication between the application and robotic devices or web services. The service layer handles all protocol specifics of the devices and services. In the application layer the relation between the devices and services is defined.

In the proposed platform, all different devices and web services are generalised to the shape of reactive services. This generalisation is very natural to (robotic) devices, but does not directly follow the service oriented model that is common for web services. A consequence of the use of reactive services is that applications can be described a very natural way, based on the inputs and outputs of the reactive services. This event based programming style is intuitive for the type of applications that are supposed to be developed on top of the platform.

The messages that are sent between services consist of plain text. Text can easily be marshalled between programming languages and operating systems, making it a compatible format. An advantage of text messages is that they can both represent data and commands, if a scripting language interpreter is used to evaluate incoming messages. By evaluating received text, it is possible to add new functionality at run time or to bring the computation to the data, resulting in lower network load and reducing response times.

7.2 Correspondence to requirements

During the beginning of the project, it was not clear which type of applications the platform should support. Although several robotic devices and web services have been considered, it was not straightforward to generalise an application description from their functionality that was more precise than the one described in Chapter 2.

The lack of an exact application description was an encouragement to make the platform as flexible as possible. This resulted in a platform where application logic can be distributed between the services or can run on a single machine as desired. Connections between services may be created and removed during execution, matching the application developers needs at any point in time.

Regarding services as being reactive, no matter how they are exactly implemented makes their usage very straightforward. An application developer will have to invest some time in
figuring out the meaning of each input and output value, but the conceptual model behind the creation of these type of applications is simple and flexible.

### 7.3 Future research

For simple applications the architecture suffices, but problems arise when connecting several applications to the same service (either a robotic device or a web service) and trying to use its resources. The variety of different robotic devices and web services guarantees a wide variety of available resources and interfaces. This type of information should be shared between the service and the connected applications, such that these applications can adapt themselves to the available resources. As an additional advantage, it would make the communicated scripts as powerful as mobile agents.

A second element of future research is the search of an appropriate security model. Currently, a security model has been implemented that is based on the Safe-Tcl model, which consists of a sandbox interpreter from which all potentially dangerous functionality has been removed. Also, the current model does not discriminate between different users of a service, granting all users the same limited amount of functionality. In the type of robotic devices and web services that have been used in the development of a prototype, this limited flexibility of the security layer didn’t cause any problems. However, a robotic device that closes the windows of your home may benefit from a different security model.

As a third area of research, an advanced visual aid may be designed and implemented. It has been mentioned several times that the way in which the platform operates can be easily visualised. There are some remarks to be made about this statement. In general, visual aids allow the programmer to define the relation between components as it is during compile time. In the proposed platform, the connections between components may change over time and depending on previous computations. This complicates the way in which the process graph is visualised and the way in which application programmers can make changes to the graph.

### 7.4 Verification

Consistency becomes a problem as soon as multiple processes have access to the same information. When this information is distributed among the nodes, as is the case for web farms, the problem becomes even more apparent. Kemme and Alonso recognised this problem and provided several protocols that promised to solve this problem.

In this thesis, one of these protocols has been modelled and evaluated. It appeared that the protocol itself was elegant and simple, but much of the underlying details and assumptions where hidden throughout their papers. These details and assumptions have been found and a model of their protocol has been described. Although the model lacks the elegance of the original protocol, it is a more precise description of the solution proposed by Kemme and Alonso.

Several desirable properties of this protocol have been described, formalised and have been evaluated. From this evaluation, the desired properties where shown to hold in the system. Although the tools are experimental and are likely to contain errors, the results intuitively match the behaviour of the protocol and the model is considered to be a correct and concise representation of the protocol.
Bibliography


BIBLIOGRAPHY


[23] Koninklijke Philips Electronics N.V. iCat research platform, 1.0 edition.


[28] A. Mathijssen. Concrete syntax for mcrl2 mu-calculus formulas and pbeses. internal communication, August 2007.


Appendix A

Scripting Languages

A.1 Tcl

The scripting language Tcl is created by John Ousterhout. In this appendix, the Tcl syntax will be described such that the examples in this thesis that are written in the Tcl scripting language can be understood. Even though the number of syntax rules of the Tcl scripting language is very limited, this chapter is not aiming at giving a complete manual to the language. This introduction to the language is roughly based on the Tcl tutorial [11] by Clif Flynt.

A.1.1 Syntax

The Tcl scripting language is all about strings. A Tcl script consists of a number of commands, separated by new lines or semicolons. A command consists of any number of words, separated by spaces. The first word of a command being the name of a procedure, all other words are parameters to this procedure. Variable values can be obtained by prefixing the variable name with a $-sign. To make things more clear, consider the following example:

```tcl
set a 10
expr $a + 5
```

This example will set the value of variable `a` to 10, will replace all references to variable values with the actual values in the second command and will then return the result of the command `expr` with the parameter `10 + 5`.

When programs become more complex, variable values might contain spaces, new lines or semicolons. In order to cope with this, two ways of grouping text are available. With double quotes, one can indicate that multiple words belong together and that substitution of escaped characters and variables should be performed. As an example:

```tcl
puts "$a + 5" ;# prints '10 + 5' to stdout
puts { $a + 5 } ;# prints '$a + 5' to stdout
```

Although any value can be passed via variables, since everything is a string, it might be more convenient to have multiple commands on the same line. To indicate a command, it can be surrounded by square brackets:

```tcl
puts [ expr $a + 10 ] ;# prints '15' to stdout
```
Scripts can be written by following these simple rules, that have been applied without exceptions. As a last example, a (somewhat inefficient) recursive procedure to print the first \( n \) integers will be given:

```tcl
proc firstN {n i {r ""}} {
  if {$i > $n} {
    puts $r
  } else {
    append r $i " "
    firstN $n [incr i] $r
  }
}
```

The exact specification of the `proc` and `for` procedures will be omitted here, since it is described in more detail in the Tcl manual. It is sufficient to observe that the first element, `proc`, takes 3 parameters with the last parameter spanning multiple lines.

### A.1.2 Language features

#### Socket communication

Tcl follows the unix approach, saying that ‘everything is a file’. This implies that writing and reading to sockets is done in the same way as it is done with local files. An event handler can be assigned to files, being executed when new information is available or the file is closed. In this way, a programmer does not constantly have to poll for new information, which may involve a blocking read operation, but can have the underlying platform inform him when there is new information available.

#### Interpreter hierarchy

Tcl scripts are evaluated, or interpreted, inside an interpreter. For Tcl, this interpreter is basically a table with all available procedures and a reference to the code or script that should be executed if the corresponding procedure is called. From within Tcl the interpreter can be queried to find out about available procedures and their implementation details. Also, `tcl` allows modifications to this table to be made during run-time.

What is making this an advanced feature is that every interpreter is capable of creating slave interpreters, having their own table of procedures and variables, which can be modified by the interpreter that created them as well. The creation of a slave interpreter, from which every potentially dangerous procedure can be removed, enables the creation of sandboxes. These sandboxes can be used to resolve the potential problems of conflicting naming of different applications and malicious scripts.

### A.1.3 Extensions

New commands can be added to the scripting language in multiple ways. They can be directly included in the script by the programmer, or included from packages that can be written both in Tk or in C/C++. This way, for every package the best balance between efficiency and ease of development can be selected. In this section, the transport layer security (TLS) extension will be introduced.
Transport Layer Security

Internet connections are prone to attacks by the outside world. Authentication and encryption are required for every non-trivial communication and the services and applications considered to be used in the architecture are no exception to this rule.

Transport Layer Security (TLS), which is the successor of the more widely known Secure Socket Layer (SSL), provides this functionality. It is a rather complex protocol, with variations as single versus mutual authentication and agreements on the encryption protocols to be used. After a connection has been established, all communication must be encrypted and deciphered appropriately.

Adding TLS to a Tcl program is fairly straightforward. A package exists which provides the functionality to create and communicate over a TLS socket. This package can be used to create a connection with a TLS server socket as follows:

```tcl
package require tls

# client's certificate
set cafile {CertificateAuthority.crt}
set certfile {certificates/john.crt}
set keyfile {certificates/john.key}

set s [tls::socket \ 
   -cafile $::cafile \ 
   -certfile $::certfile \ 
   -keyfile $::keyfile \ 
   localhost 3000]
```

For the creation of the server process, the implementation is similar. After being created, `s` can be used like any file handle, as explained in section A.1.2. Note that there are three different files used which contain certificate keys. Their content is based on the TLS protocol, which is too complex to explain here. The files themselves can easily be created with a tool named SimpleCA.¹

---

¹The SimpleCA tool can be downloaded from http://www.vpnc.org/SimpleCA/
Appendix B

Verification

B.1 The model

sort
  Node = struct N1 | N2;
  Result = struct commit | abort;
  Action = struct r | w;
  Phase = struct read | send | lock | committed | aborted;
  Transaction = struct t1 | t2;
  TransactionItem = struct tr(identifier:Transaction, phase:Phase);
  TransactionQueue = struct tq(node:Node, transaction:TransactionItem);

act
  % Communication node with user
  rUQ, sUQ, cUQ:Node#Transaction;
  rUR, sUR, cUR:Node#Transaction#Phase;

  % Communication node with network
  rSend, sSend, cSend: Node#Transaction;
  rSend, sSend, cSend: Node#Transaction#Result;
  rReceive, sReceive, cReceive: Node#Transaction;
  rReceive, sReceive, cReceive: Node#Transaction#Result;

  % Communication with the lock manager
  rLock, sLock, cLock : Node#Transaction#Action;
  rLockResult, sLockResult, cLockResult : Node#Transaction#Result;
  rLockDecision, sLockDecision, cLockDecision : Node#Transaction#Result;
  rRejectLock, sRejectLock, cRejectLock : Node#Transaction;
  rLockResultWS, sLockResultWS, cLockResultWS : Node#Transaction#Result;

  % Internal decisions of the lock manager
  NoLocks, NoConflictingLocks, ConflictingLocks:Node#Transaction;
map filter : List(TransactionQueue)#Node -> List(TransactionQueue);
remove : List(TransactionQueue)#TransactionQueue -> List(TransactionQueue);
remove : List(TransactionItem)#TransactionItem -> List(TransactionItem);
update : List(TransactionItem)#TransactionItem#Phase -> List(TransactionItem);

var
LTQ : List(TransactionQueue);
LTI : List(TransactionItem);
n: Node;
t: TransactionItem;
tq: TransactionQueue;
p: Phase;

eqn
(#LTQ == 0) -> remove(LTQ, tq) = [];
(#LTQ != 0) -> remove(LTQ, tq) = if (identifier(transaction(head(LTQ))) == identifier(transaction(tq)) &&
node(head(LTQ)) == node(tq),
remove(tail(LTQ),tq) ,
[head(LTQ)] ++ remove(tail(LTQ),tq)
);

(#LTI == 0) -> remove(LTI, t) = [];
(#LTI != 0) -> remove(LTI, t) = if (identifier(head(LTI)) == identifier(t),
remove(tail(LTI),t) ,
[head(LTI)] ++ remove(tail(LTI),t)
);

(#LTQ == 0) -> filter(LTQ, n) = [];
(#LTQ != 0) -> filter(LTQ, n) = if(n == node(head(LTQ)),
[head(LTQ)] ++ filter(tail(LTQ),n),
filter(tail(LTQ),n)
);

(#LTI == 0) -> update(LTI, t, p) = [];
(#LTI != 0) -> update(LTI, t, p) = if (identifier(head(LTI)) == identifier(t) ,
[tr(identifier(head(LTI)), p)] ++ update(tail(LTI), t, p) ,
[head(LTI)] ++ update(tail(LTI), t, p)
);

proc USER(t:List(Transaction)) =
(#t > 0)
-> sum n:Node . SUQ(n, head(t)) . USER(tail(t))
<> delta;

proc NETWORK(transactions: List(TransactionQueue),
B.1. THE MODEL

results: List(TransactionQueue)) =

sum n:Node . (
% Receive inputs from users
sum t:Transaction . (rSend(n,t) . NETWORK(transactions ++ [tq(N1,tr(t,lock)),
  tq(N2,tr(t,lock))], results)
+
sum r:Result . rSend(n,t,r) . (r == abort)
  -> NETWORK(transactions, results ++ [tq(N1,tr(t,aborted)),
    tq(N2,tr(t,aborted))])
<> NETWORK(transactions, results ++ [tq(N1,tr(t,committed)),
    tq(N2,tr(t,committed))])
)
)
+

% Multicast the transaction messages
(#filter(transactions, n) != 0)
-> sReceive(n, identifier(transaction(head(filter(transactions, n))))) .
  NETWORK(remove(transactions, head(filter(transactions, n))), results)
<> delta
+

% Multicast the results (in arbitrary order)
sum t:Transaction . sum p:Phase . (tq(n,tr(t,p)) in results)
  -> ( (p == committed)
    -> sReceive(n,t,commit)
    <> sReceive(n,t,abort)
  ) . NETWORK(transactions, remove(results, tq(n,tr(t,p))))
 <> delta
);

proc NODE(n:Node) =
  NODE_TRANSACTION_MANAGER(n,[]) ||
  NODE_COMMUNICATION_MANAGER(n) ||
  NODE_LOCK_MANAGER(n,[],[],[]);
sum t:Transaction .
  (tr(t,read) in local)
  -> sLock(n,t,r) .
  sum r:Result .
    rLockResult(n, t, r) .
    NODE_TRANSACTION_MANAGER(n, update(local, tr(t,read), send))
<> delta

+ % Send the lock request for all write operations to the GCM
sum t:Transaction .
  (tr(t,send) in local)
  -> ssSend(n,t) .
  NODE_TRANSACTION_MANAGER(n, update(local, tr(t,send), lock))
<> delta

+ % Listen for request for abortion by the lock manager
sum t:Transaction .
  rRejectLock(n,t) . ( 
    % The lock can be the lock on a message that is (not yet) send:
    (tr(t,send) in local)
    -> sLockDecision(n,t,abort) .
  sClearLock(n,t) .
  sUR(n,t,aborted) .
  NODE_TRANSACTION_MANAGER(n, remove(local, tr(t,read)))
<> delta
+
  (tr(t,lock) in local)
  -> ssSend(n,t,abort) .
  sUR(n,t,aborted) .
  NODE_TRANSACTION_MANAGER(n, remove(local, tr(t,lock)))
<> delta
)

+ % Listen for results of write set lock requests:
sum t:Transaction .
  sum r:Result .
  rLockResultWS(n,t,r) . ( 
    (tr(t,lock) in local)
    -> ((r == abort) ->
      sUR(n,t,aborted)
    <> sur(n,t,committed)
  )
  . ssSend(n,t,r) .
  NODE_TRANSACTION_MANAGER(n, remove(local, tr(t, lock)))
<> NODE_TRANSACTION_MANAGER(n, local)
);
proc NODE_COMMUNICATION_MANAGER(n:Node) =
  sum t:Transaction .
    rReceive(n, t) .
    sLock(n, t, w) .
  sum r:Result .
    rReceive(n, t, r) .
    sLockResult(n, t, r) .
    sLockResultWS(n,t,r) .
    NODE_COMMUNICATION_MANAGER(n)
  +
  sum t:Transaction .
    sum r:Result .
    rReceive(n, t, r) .
    sLockDecision(n,t,r) .
  NODE_COMMUNICATION_MANAGER(n);

map
  remove: List(Transaction)#Transaction -> List(Transaction);
var
  LTI : List(Transaction);
  t: Transaction;
  eqn
    (#LTI == 0) -> remove(LTI,t) = [];
    (#LTI != 0) -> remove(LTI,t) = if ( head(LTI) == t , remove(tail(LTI),t)
       , [head(LTI)] ++ remove(tail(LTI),t));

act removeLocks : Node#Transaction;
  rClearLock, sClearLock, cClearLock:Node#Transaction;

proc NODE_LOCK_MANAGER(n:Node,
  readLocks:List(Transaction),
  writeLocks:List(Transaction),
  releases:List(Transaction)
) =
  sum t:Transaction . (  
    sum a:Action .
      rLock(n,t,a) . (  
        (a == r) -> sLockResult(n,t,commit) . NODE_LOCK_MANAGER(n,
          readLocks ++ [t],writeLocks,releases)
        <> (  
          NoLocks(n,t) .
          sLockResult(n,t,commit) .
          NODE_LOCK_MANAGER(n, remove(readLocks,t),
            remove(writeLocks,t) ++ [t], releases)
        )
      )
  )
  NODE_LOCK_MANAGER(n, remove(readLocks,t),
    remove(writeLocks,t) ++ [t], releases) +
  NoConflictingLocks(n,t).
APPENDIX B. VERIFICATION

sLockResult(n, t, commit)
NODE_LOCK_MANAGER(n, remove(readLocks, t),
    remove(writeLocks, t) ++ [t], releases)
+ (#remove(readLocks, t) != 0)
-> ConflictingLocks(n, t)
    sRejectLock(n, head(remove(readLocks, t)))
    sLockResult(n, t, commit)
    NODE_LOCK_MANAGER(n, remove(readLocks, t),
        remove(writeLocks, t) ++ [t], releases)
<> delta

+ sum r:Result.
    rLockDecision(n, t, r)
    NODE_LOCK_MANAGER(n, readLocks, writeLocks, releases ++ [t])
+ rClearLock(n, t). NODE_LOCK_MANAGER(n,
    remove(readLocks, t),
    remove(writeLocks, t),
    remove(releases, t)
)+
(t in writeLocks && t in releases) ->
    removeLocks(n, t). NODE_LOCK_MANAGER(n,
    remove(readLocks, t),
    remove(writeLocks, t),
    remove(releases, t)
)<> delta

init
allow({
cUQ,
sUR,
cSend,
cReceive,
cLockResultWS,
cLock,
cLockResult,
cLockDecision,
cRejectLock,
NoLocks, NoConflictingLocks, ConflictingLocks, removeLocks, cClearLock
})
,comm({

```
B.2 Requirements

B.2.1 Req 1.1

\[ \forall X. (\mathbf{true}) \land \forall n: \text{Node}. \forall t: \text{Transaction}. \exists n_2: \text{Node}. \lnot \mathbf{true} \]

B.2.2 Req 1.2

\[ \forall X. (\mathbf{true}) \land \forall n: \text{Node}. \exists t: \text{Transaction}. \exists n': \text{Node}. \exists p: \text{Phase}. \lnot \mathbf{true} \]

B.2.3 Req 1.3

\[ \forall X. (\mathbf{true}) \land \forall n: \text{Node}. \forall t: \text{Transaction}. \forall a: \text{Action}. \exists n': \text{Node}. \exists p: \text{Phase}. \lnot \mathbf{true} \]

B.2.4 Req 2.1 and req 2.2

\[ \forall X. (\mathbf{true}) \land \forall n: \text{Node}. \forall t: \text{Transaction}. \lnot \mathbf{true} \]
APPENDIX B. VERIFICATION

\[
\begin{align*}
\mu Y. \\
[!\exists p: \text{Phase} . \text{sUR}(n,t,p)] Y &\& <\text{true}>true
\end{align*}
\]

And the additional formula for requirement req 2.2.

\[
\begin{align*}
\nu X. [\text{true}]X &\& \\
\forall n: \text{Node}. \\
\forall t: \text{Transaction}. [\text{cUQ}(n,t)]( \\
(\mu Y. <\text{true}>true &\& \\
[!\exists p: \text{Phase} . \text{sUR}(n,t,p)] Y &\& \\
[\exists p: \text{Phase} . \text{sUR}(n,t,p)]( \\
\nu Z. [\text{true}]Z &\& [\exists p: \text{Phase} . \text{sUR}(n,t,p)]false
\end{align*}
\]

B.2.5 Req 2.3 and req 2.4

\[
\begin{align*}
\forall m: \text{Node}. ( \\
\nu X. [\text{true}]X &\& \\
\forall n: \text{Node}. \\
\forall t: \text{Transaction}. \\
\forall r: \text{Result}. [\text{cSend}(n,t,r)]( \mu Y. <\text{true}>true &\& ![\text{removeLocks}(m,t,r)]Y)
\end{align*}
\]

B.2.6 Req 2.5

\[
\begin{align*}
\nu X. [\text{true}]X &\& \\
\forall m: \text{Node}. \\
\forall t: \text{Transaction}. [\text{cUQ}(m,t)]( \\
\forall n: \text{Node}. ( \\
\text{val}(n \neq m) \Rightarrow \nu Y. [\exists r: \text{Result}. \text{cSend}(n,t,r)]false
\end{align*}
\]

B.2.7 Req 3.1

\[
\begin{align*}
\nu X(1: \text{List(Transaction)} = [], m: \text{List(Transaction)}= []). \\
\forall t: \text{Transaction}. ( \\
[\text{cLockW}(N1,t)] (\exists k: \text{List(Transaction)}. ( \\
\end{align*}
\]
B.2. REQUIREMENTS

\[
\text{val}(k ++ l == m || l == k ++ m) \land X([t] ++ l, m)
\]

\[
\land \quad \text{cLockW}(N2,t) \quad (\exists c:\text{List(Transaction)}. \quad \text{val}(c ++ l == m || l == c ++ m) \land X(1, [t]++ m))
\]

\[
\land \quad \text{!exists n:Node. exists tt:Transaction. cLockW(n,tt) X(1,m)}
\]

B.2.8 Req 3.2

\[
\forall n:Node. \quad (\nu X. \quad [true]X \land \quad \forall m:Node. \quad \forall t:Transaction. \quad [cSend(m,t)](\mu Y. \quad <true> \land \quad [!cLock(n, t,w)]Y)
\]