DEVELOPING CORRECT DISTRIBUTED SYSTEMS: REAL WORLD CASE STUDIES

Helgi Leifsson

Computer Science B.Sc.

2010

Author: Helgi Leifsson
SSN: 121176-5489
Instructor: Marjan Sirjani

Tölvunarfræðideild
School of Computer Science

Útdráttur

Smíði áreiðanlegra hugbúnaðarkerfa er flókin en samt sem áður mikilvæg áskorun nútíma hugbúnaðarverkfræði. Grundvallaratriði til að ná fram áreiðanleika er aðferðafræðin sem beitt er við að þróa og sannreyna bæði hannanir og útfærslur. Það leikur enginn vafi á því að eitt af grundvallarverkefnum tölvunar- og upplýsingafræða er áframhaldandi framþróun í aðferðum hugbúnaðarþróunar: Við þurfum betri tæki og tól til þróunar áreiðanlegra og fyrirsjáanlegra hugbúnaðarkerfa.

Á meðan upplýsingakerfi verða sífellt mikilvægari í samfélagi okkar, þá fjölgar dreifðum, ólíkum kerfum ört. Dreifð kerfi samanstanda af mörgum samvinnandi einingum sem oftast eru samanþjappaðar eða dreifðar yfir netkerfi og eiga samskipti á ósamstilltan hátt.

Í þessu verkefni vinnum við innan viðameira verkefnis sem miðar að því að sýna fram á nýstárlegar hugmyndir, aðferðir og verklag við að þróa áreiðanleg og traust dreifð og ósamstillt kerfi.

Með því að vinna með tiltekið tilfelli hugbúnaðarkerfa munum við kynnast áskorunum líkanagerðar og sannreyningar þeirra með Rebeca líkanasmiðnum. Við getum þá flokkað kosti og ókosti hverrar nálgunar fyrir sig sem mun leiða okkur að betri aðferðum og verklagi.

Í þessu verkefni hönnum við, gerum líkan af og sannreynum dreifðan orrustuskipaleik sem dæmi um ferlið við að þróa viðeigandi dreift kerfi og metum áhrif þess miðað við að beitast við prófanir eingöngu. Niðurstöður þess eru þær að án líkanagerðar í því tilfelli, hefðu villur auðveldlega getað sloppið í gegn.

Abstract

Building reliable software systems is a complex but important challenge of modern engineering. A fundamental determiner of software reliability is the methodology used to develop and verify both designs and implementations. There is no question that one of the fundamental tasks in computer and information science is advancing the state of our development methods: We need better techniques and tools for developing correct and predictable software systems.

As information networks are becoming increasingly important in our society, the number of distributed heterogeneous software systems is rapidly growing. Distributed systems consist of multiple cooperating components where the components are typically encapsulated systems or objects spread over a network, interacting via asynchronous communication.

In this project we work within a broader project which aims towards establishing novel ideas, methods, and techniques for developing reliable and trustworthy distributed and asynchronous systems.

By working on specific case studies, we will find out the challenges in modeling and verifying such systems using the Rebeca model checker. We can then classify the advantages and disadvantages of different approaches which will lead us to better methods and techniques.

In this project we design, model and verify a distributed battleship game as an example of the process of developing correct distributed systems and evaluate it‘s effectiveness vs. using testing alone. Results of which are, that in that particular case study, without modeling, bugs might easily have slipped through.

Contents

Table of Figures

1. Concurrency Challenges

The challenges with concurrency are both varied and many but they all start with the race condition [1]. It arises in software when separate processes or threads of execution are using some type of shared memory. As an example, consider two threads that access a critical section, put a value into register, increment by one and write back into the critical section. Sequentially the result is two but concurrently the result is one as shown in Fig. 1-1 to 1-3.

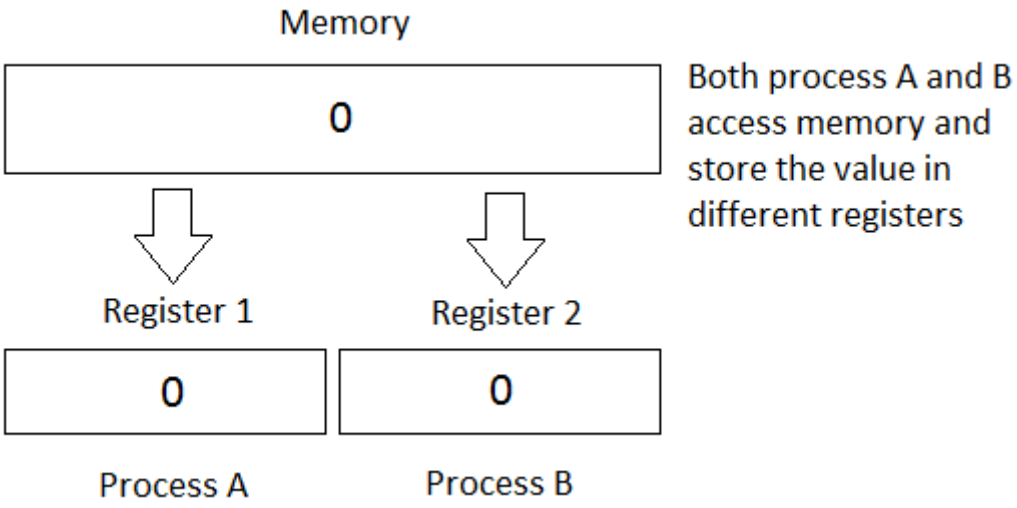

Figure 1-1. Race Condition

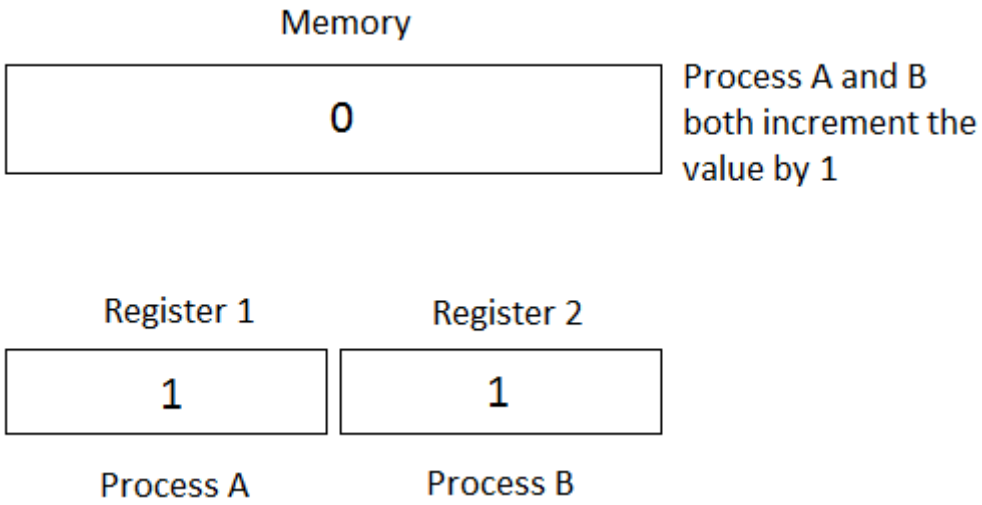

Figure 1-2. Race Condition

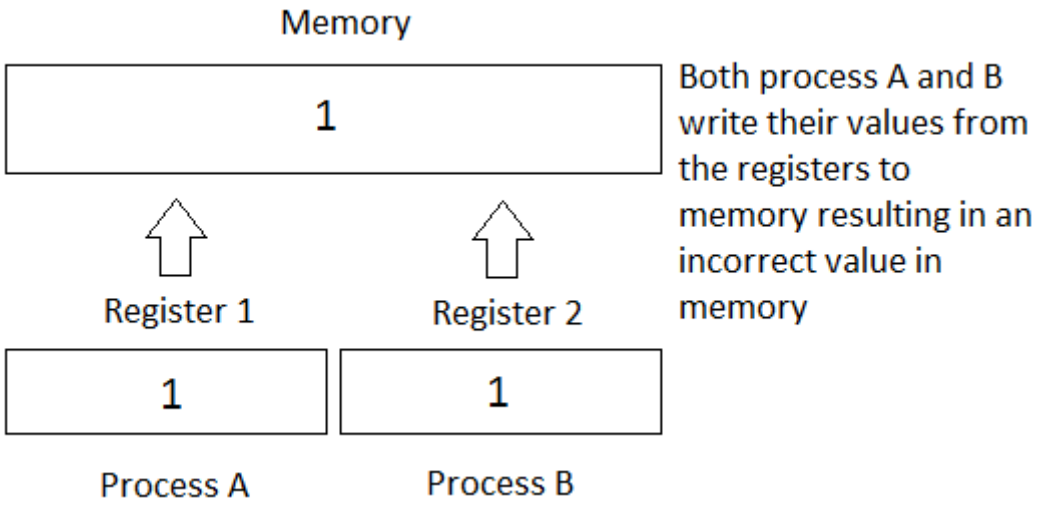

Figure 1-3. Race Condition

Problem with this is that, however unlikely, if it happens, it‘s unreproducable. This example could be a lot of other concurrent systems as well. It could be two clients working on a bank account, a flight reservation system or players playing an online game. Same underlying principles still apply and the same concurrency problems arise.

1.1 Mutual exclusion

One way of resolving this is with mutual exclusion which „locks“ the shared memory while it‘s being used by a process [1].

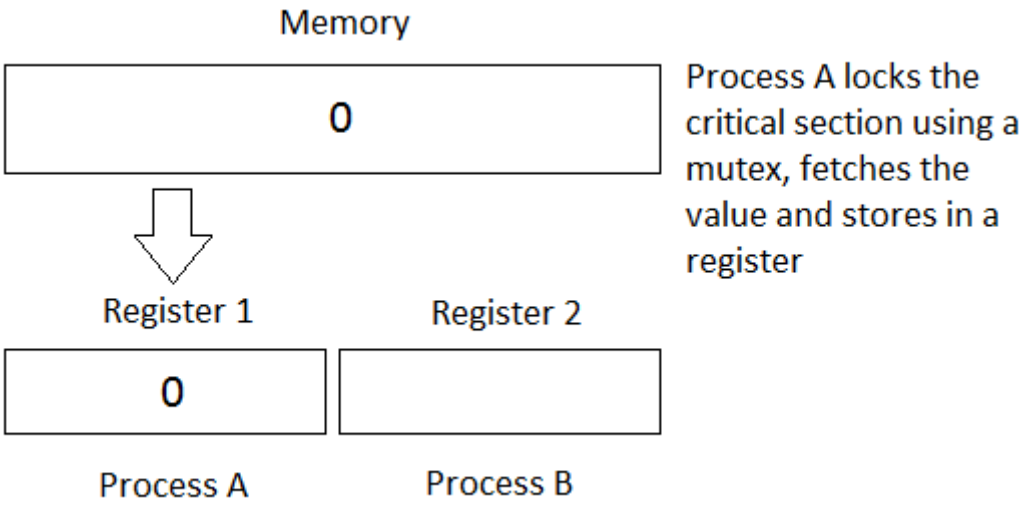

Figure 2-1. Mutual Exclusion

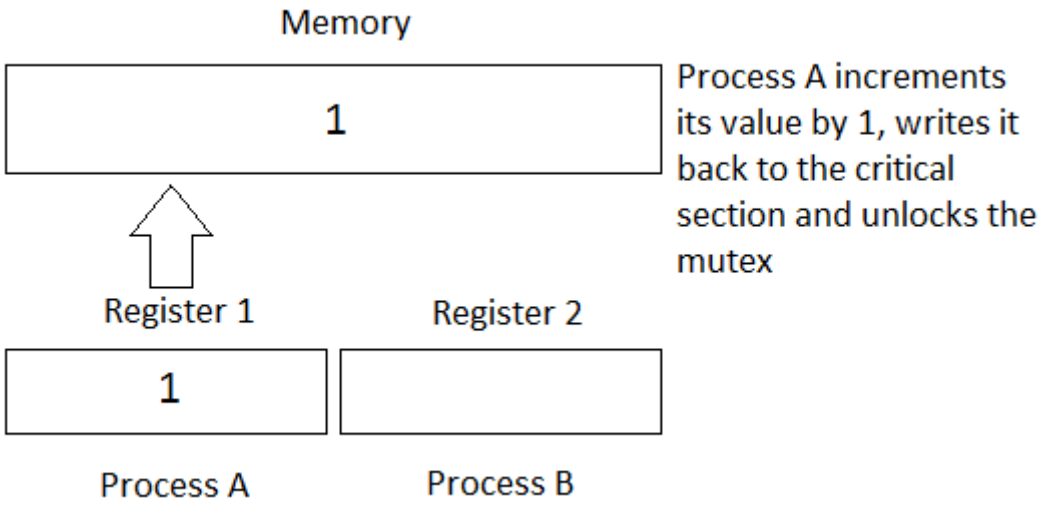

Figure 2-2. Mutual Exclusion

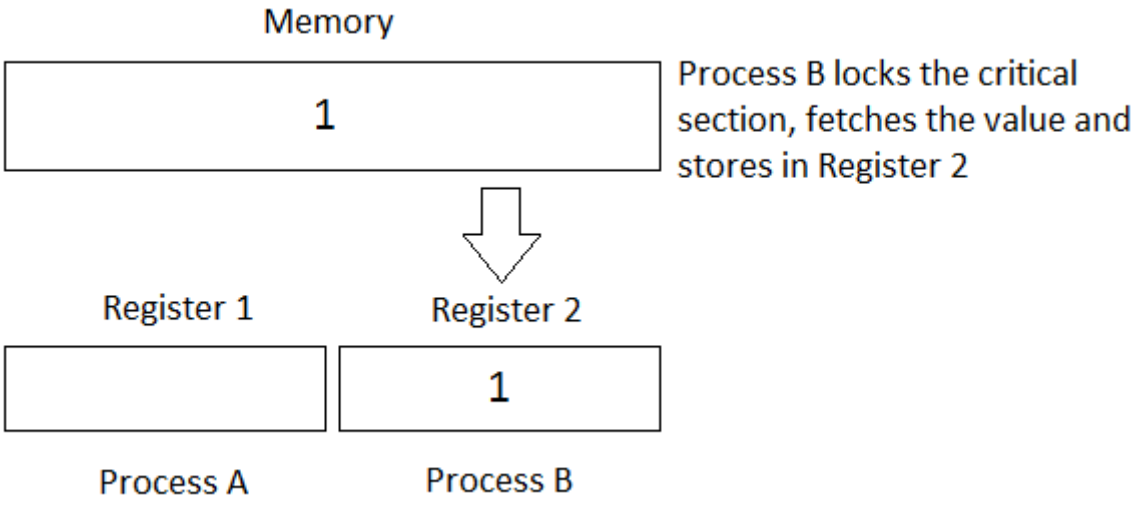

Figure 2-3. Mutual Exclusion

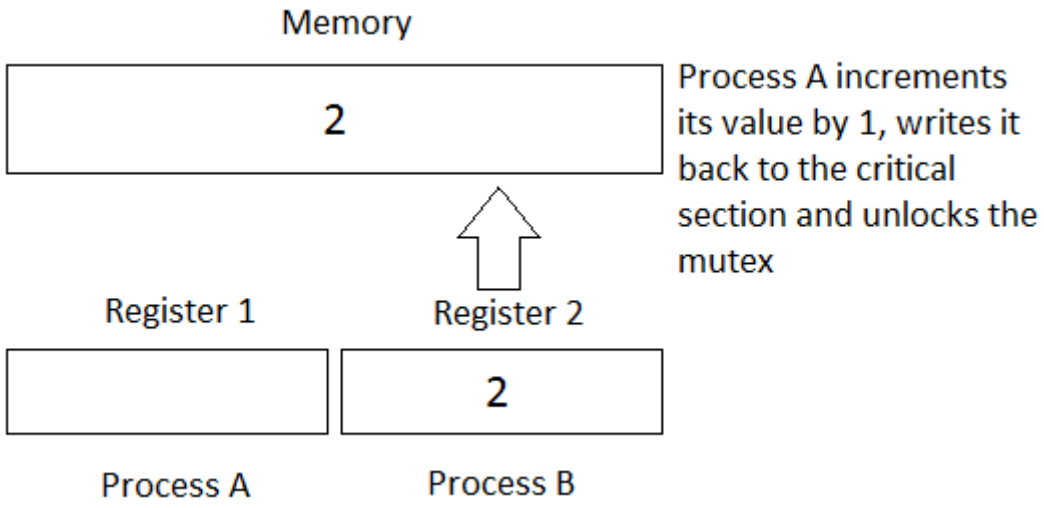

Figure 2-4. Mutual Exclusion

1.2 Deadlock

Mutual exclusion introduces deadlocks wherein two or more competing actions are each waiting for the other to finish and thus neither ever does. A real world example would be something like two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone. Figures 3-1 to 3-2 show an example with memory and processes.

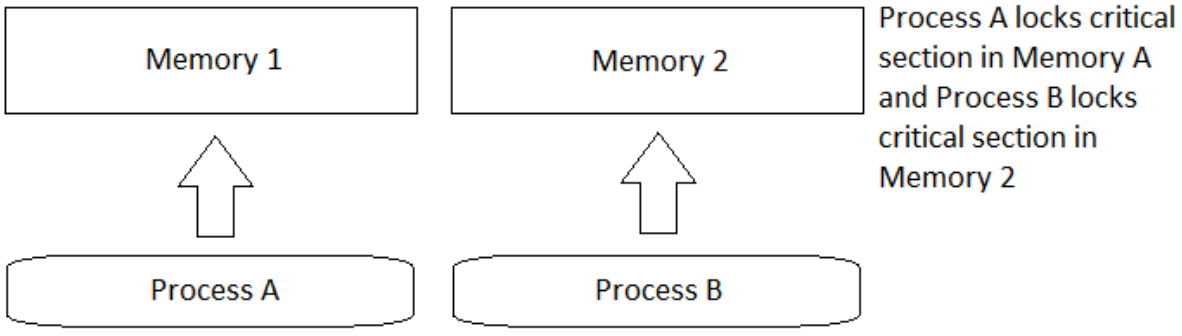

Figure 3-1. Deadlock

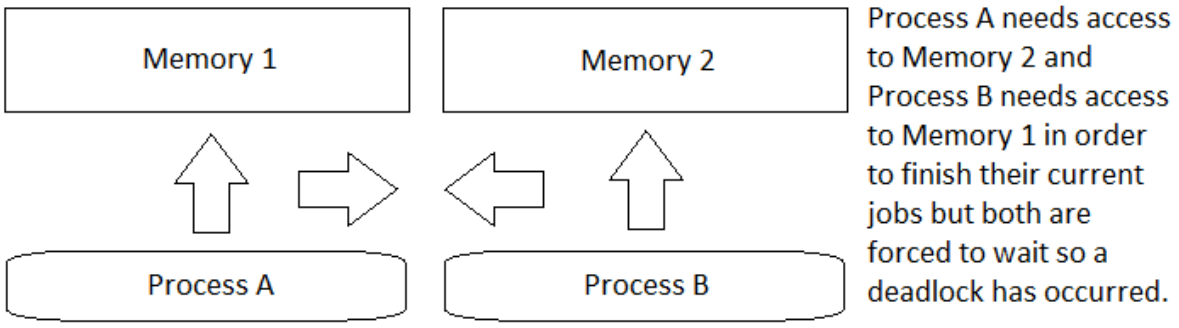

Figure 3-2. Deadlock

To summarize: having processes use shared memory leads to race conditions. To work around race conditions, we use locks which can lead to deadlocks.

To get around these problems, one way is to avoid sharing writable memory and use message passing to communicate between threads instead.

1.3 Alternative to Shared Memory: Message Passing

In a message passing system there‘s no shared memory so the processes communicate by sending each other messages. The messages are then stored in a message buffer or a queue to be later picked up and processed. This means that if a message queue fills up, a message may be dropped and the system deadlocks.

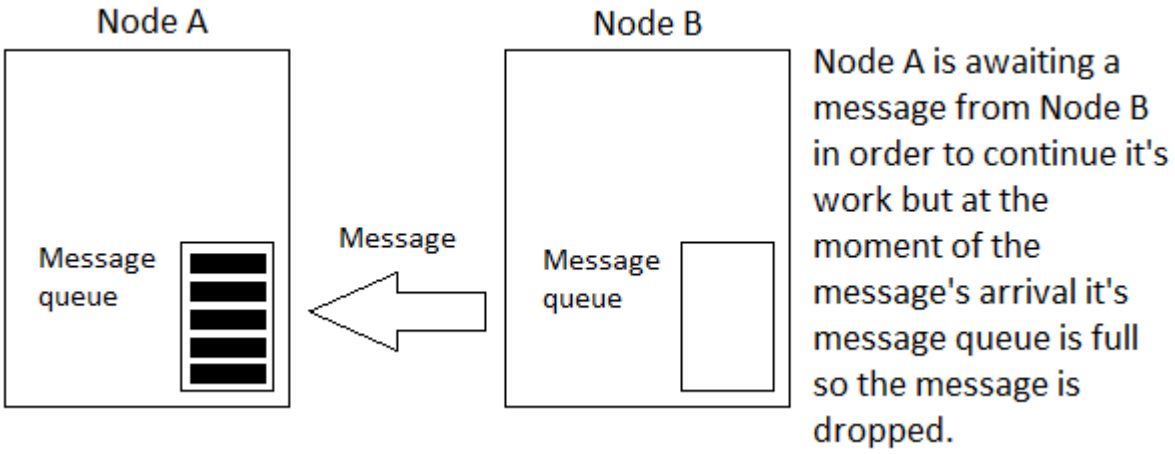

Figure 4-1. Queue Overflow

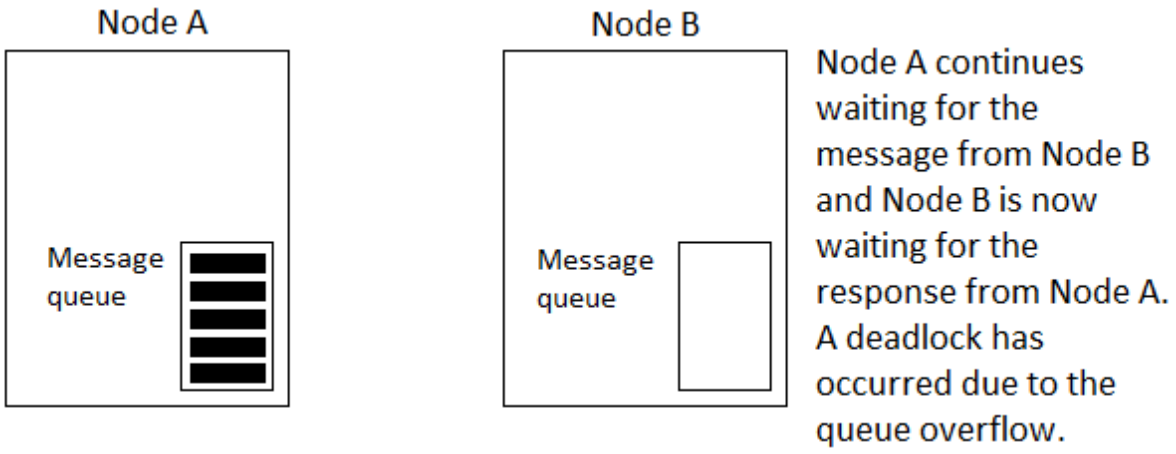

Figure 4-2. Queue Overflow

Network reliability is also a problem as a message might get lost on the network. In that case we have processes time when they send a message and retransmit if they don‘t receive an acknowledgement back [2]. Despite that there are also possible issues with the message passing design itself irrespective of the network layer, which we resolve through modeling as we shall see.

Sidenote: message passing tends to be less efficient than using threads. One way might be to limit the message sizes and doing message passing using the register [8].

2. Modeling

Scientific modeling is used in a number of different disciplines and usually revolves around generating abstract, conceptual, graphical and/or mathematical models. In software we can use model checking as a technique for automatically verifying the correctness of finite-state systems. If it is either impractical or impossible to create experimental conditions to directly measure an outcome, we can create a model and have a model checker automatically test whether it meets a given specification. Benefits of which are reducing the number of defects in the system, facilitating early evaluation of the system, capturing and organizing the understanding of the system, permit early exploration of alternatives and increasing the decomposition and modularization as we will see.

As opposed to a simulation, which means simulating the whole system operating, in modeling we model only a specific part of operations or a certain mechanism each time.

For this, we can use a modeling language which is an artificial language that captures the components, information, structure and rules within a system that have to do with what‘s being developed, tested, or verified and leave out any irrelevant details.

2.1 Rebeca Model Checker

In this project we use the Rebeca Model Checker which is actor-based and uses an asynchronous message-driven object-based computational model which makes it ideal for finding deadlocks and queue overflows. It has a Java-like syntax and is relatively easy to use.

The model checker creates all the possible states the model can be in and checks each state whether there‘s a deadlock or a queue overflow. The process used to achieve that is abstracting all the elements of the system that have to do with the message passing layer, model and verify before adding all the details of the application layer or the programming part.

The model checker has no concept of time so it has the objects take turns in processing messages. An object can send multiple messages but only process one each turn. This reduces the state space and makes the model simpler without sacrificing correctness [5].

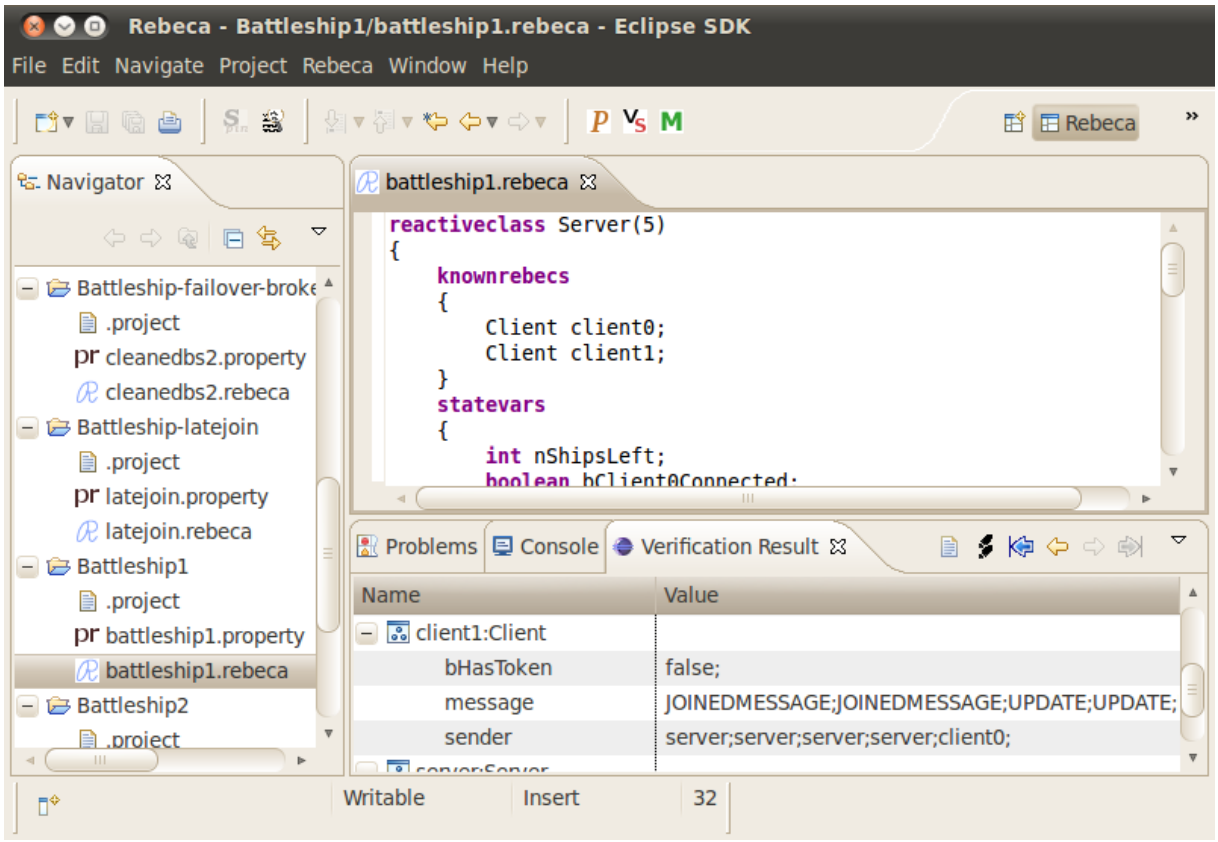

Figure 5. shows the Rebeca Model Checker

The Rebeca model checker in action. Left shows different projects, upper right is actual code and bottom are the verification results.

2.2 Basic Rebeca Model Layout

A Rebeca model consists of reactive classes or rebecs which contain information on known rebecs, its state variables and an initial function that is used to initialize a rebec. Following that we set up message servers, which are functions the rebecs use to communicate with each other. Once a certain type of message is received, the approriate message server is activated. The methods are remotely invoked but it‘s also possible for a rebec to invoke one by itself. This does however require it to send a message to itself which goes into the message queue.

After the reactive classes code comes the main function where their instances are created and initialized with their respective known rebecs as parameters.

3. Case study: Distributed Battleship Game

In this case study we model a distributed battleship game with a failover node that can take over in case the server goes down and is also capable of handling late joins. The game works as such: players, two or more, connect to a server and join a game with a single gameboard. On the gameboard ships have been placed in a grid and the object of the players is to take turns shooting at the ships in an attempt to sink as many as possible.

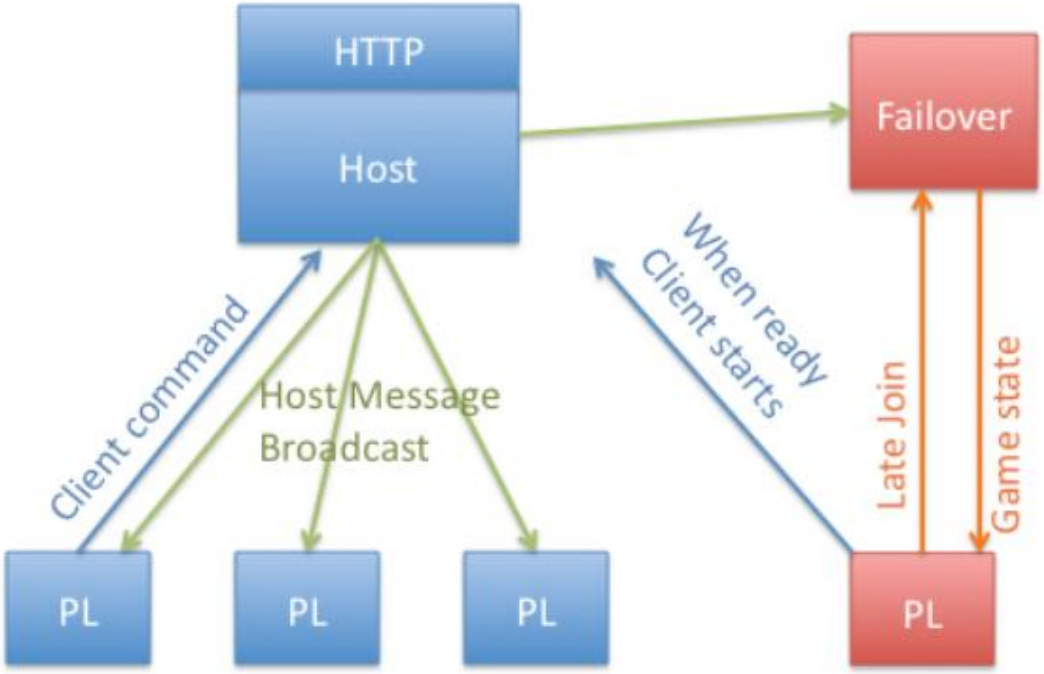

Figure 6. Distributed Battleship Game Network Architecture

3.1 Distributed Systems Mutual Exclusion Algorithms

For this to work we need some kind of mutual exclusion or the clients will have to face the race condition. In distributed systems three mutual exclusion algorithms have been extensively studied. A centralized, distributed and a token ring [2].

In a centralized algorithm the clients would send requests to access the critical region, wait for response, and then access if possible. The server would be responsible for knowing who‘s turn it is to shoot and grant access based on that. This system would however require the clients to constantly poll the server for access which would create

considerable overhead in message passing. In addition, as the system grows and more clients are added, the server might become a performance bottleneck.

We could also use a distributed algorithm but then we'd have to introduce certain unnecessary complexities such as logical clocks and having all the participants vote on who gets access. This would require everyone to stay up to date on who's turn it is, which increases coupling, and multiple messages being sent across the system to everyone else which introduces overhead. Adding or removing clients would need updating everyone participating on the changed state of the system so scaling would be a complex problem.

Third algorithm is a token ring system where each client only needs to know who's next in line behind him and when done with his turn passes the „token" to that player. The player receiving the token then takes his turn and passes the token along and so forth. This way we have less coupling and have decreased the amount of actual messages needed within the system. Adding or removing clients also scales better for now only one of the clients needs to be updated on the order in which the players take turn.

3.2 Token Ring Model

To verify a token ring system we need to figure out the systems circular interactions protocol or the dataflow [2][3]. In this case it is a series of events where messages are being passed:

- server broadcasts a new game available
- clients join
- clients shoot
- clients pass the token
- server sends a game over
- new game is available again.

What's important about these messages is that if any one of them is lost, the system deadlocks. This could be due to queue overflows or other reasons.

3.2.1 Rebecs used

This simple model contains a server and two clients. The server rebec has two known other rebecs, client0 and client1, and three state variables: how many ships are left, and whether the clients have connected or not.

Initially we set the value of ships left as 5 (for good measure), both clients have not connected so their values are false and broadcast a new game available message. The message causes both clients to connect by sending a join message to the server. When the server receives a join message it sends a message to the clients already connected that a join has taken place. No further responses lead on from that point.

Once both clients have joined, the server broadcasts the current gamestate and passes the token to client0 in this case. Client0 then sends a shoot message to the server and a token message to the next client. The server responds to the shoot messages by broadcasting the updated gamestate to the clients and once all ships have been sunk a gameover message is sent. This cycle should then repeat itself with the server sending a new game available message.

Code Example 1: Server and Two Clients

```
reactiveclass Server(5)
{
        knownrebecs
        {
                Client client0;
                Client client1;
        }
        statevars
        {
                int nShipsLeft;
                boolean bClient0Connected;
                boolean bClient1Connected;
        }
        msgsrv initial()
        {
                nShipsLeft = 5;
                bClient0Connected = false;
                bClient1Connected = false;
                self.NewGameAvailable();
        }
        msgsrv Join()
        {
                if( sender == client0 )
                {
                        bClient0Connected = true;
                        if( bClient1Connected == true ) client1.JoinedMessage();
                }
                else if( sender == client1 )
                {
                        bClient1Connected = true;
                        if( bClient0Connected == true ) client0.JoinedMessage();
```

```
                }
                if( bClient0Connected == true && bClient1Connected == true )
                {
                        client0.Update();
                        client1.Update();
                        client0.HasToken();
                }
        }
        msgsrv ServerShoot()
        {
                nShipsLeft = nShipsLeft - 1;
                if( nShipsLeft == 0 )
                {
                        client0.GameOver();
                        client1.GameOver();
                        self.initial();
                }
                else
                {
                        client0.Update();
                        client1.Update();
                }
        }
        msgsrv NewGameAvailable()
        {
                client0.Connect();
                client1.Connect();
        }
}

reactiveclass Client(5)
{
        knownrebecs
        {
                Server server;
                Client clientNext;
        }
        statevars
        {
        }
        msgsrv initial()
        {
        }
        msgsrv Connect()
        {
                server.Join();
        }
        msgsrv Update()
        {
        }
        msgsrv HasToken()
        {
                server.ServerShoot();
                clientNext.HasToken();
        }
        msgsrv GameOver()
        {
                self.initial();
        }
        msgsrv JoinedMessage()
        {
        }
}

main
{
        Server server( client0, client1 ):();
        Client client0( server, client1 ):();
        Client client1( server, client0 ):();
}
```

Example of Bug

Bug found by the model checker at this stage of the system was a queue overflow that could have been easily missed without using a model checker. If we are supposed to make a system where a client shoots and passes a token it would seem intuitive to have an operation that simply sends the shoot message to the server and simultaneously passes the token to the next client. But as we'll see, it could lead to a deadlock.

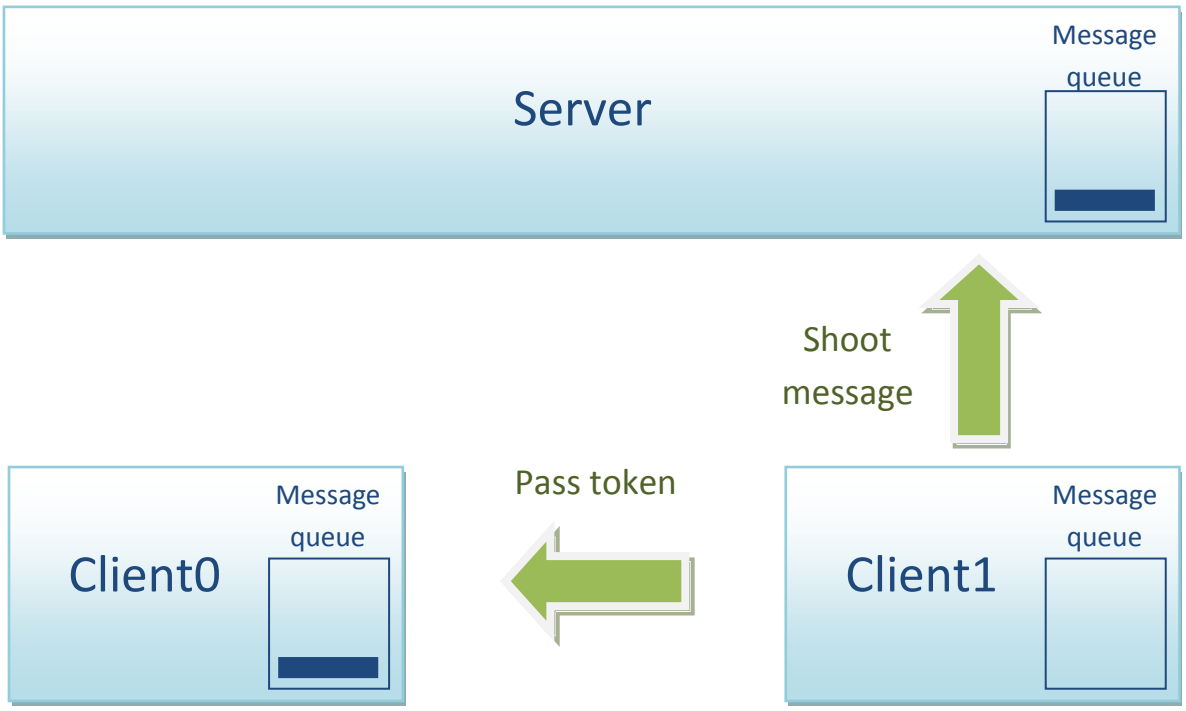

Figure 7-1. Distributed Battleship Game Queue Overflow

In the beginning client1 sends a shoot message to the server and a pass token message to the other client leaving the messages in their queues.

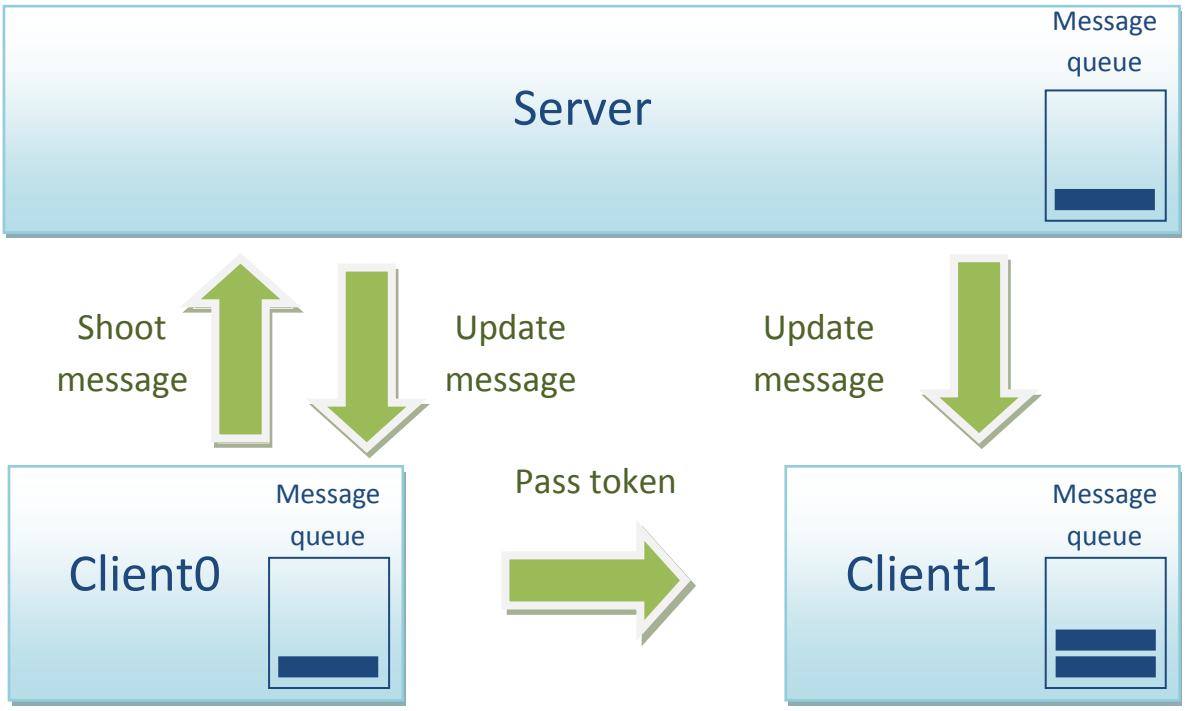

Figure 7-2. Distributed Battleship Game Queue Overflow

Immediately after receiving their respective messages the server multicasts messages to the clients with an updated gameboard. The client that previously received the token now sends a shoot message and passes the token along, which as is seen in Client1's message queue leads him to receive two messages simultaneously from the server and from Client0.

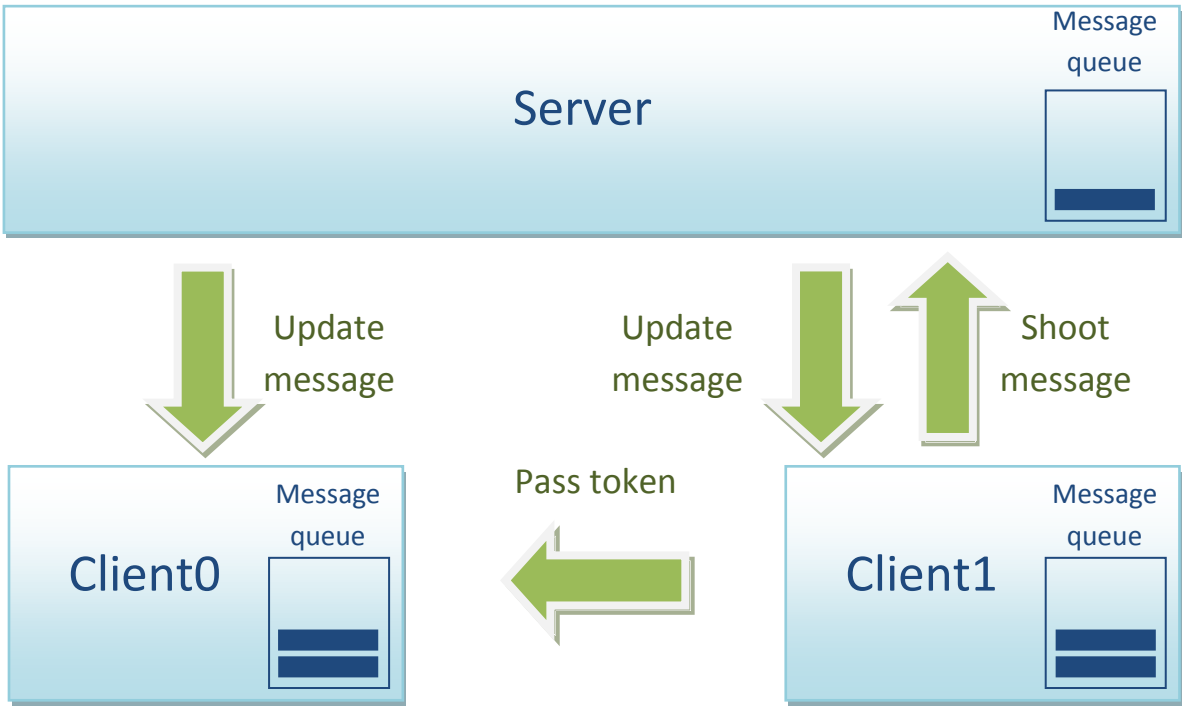

Figure 7-3. Distributed Battleship Game Queue Overflow

Client1 shoots and passes the token as before, receives an update message from the server but can only process one message at a time so at this rate, the client's message queues start to slowly fill up.

Bugs resolved

Once a message queue fills up, one of the messages that effect the circular wait condition [3] can be dropped leading to a deadlock. Also, if the client shoots before receiving the update message from the server, he will be using an old gamestate and won't be making proper use of mutual exclusion, leading to a race condition. That means to have the client wait for the update message before shooting and passing but a better way is to have the client wait after shooting and then passing the token. That way if the shoot message is lost due to network or other problems the update message can serve as an „acknowledgement" or „ack" message increasing the systems reliability.

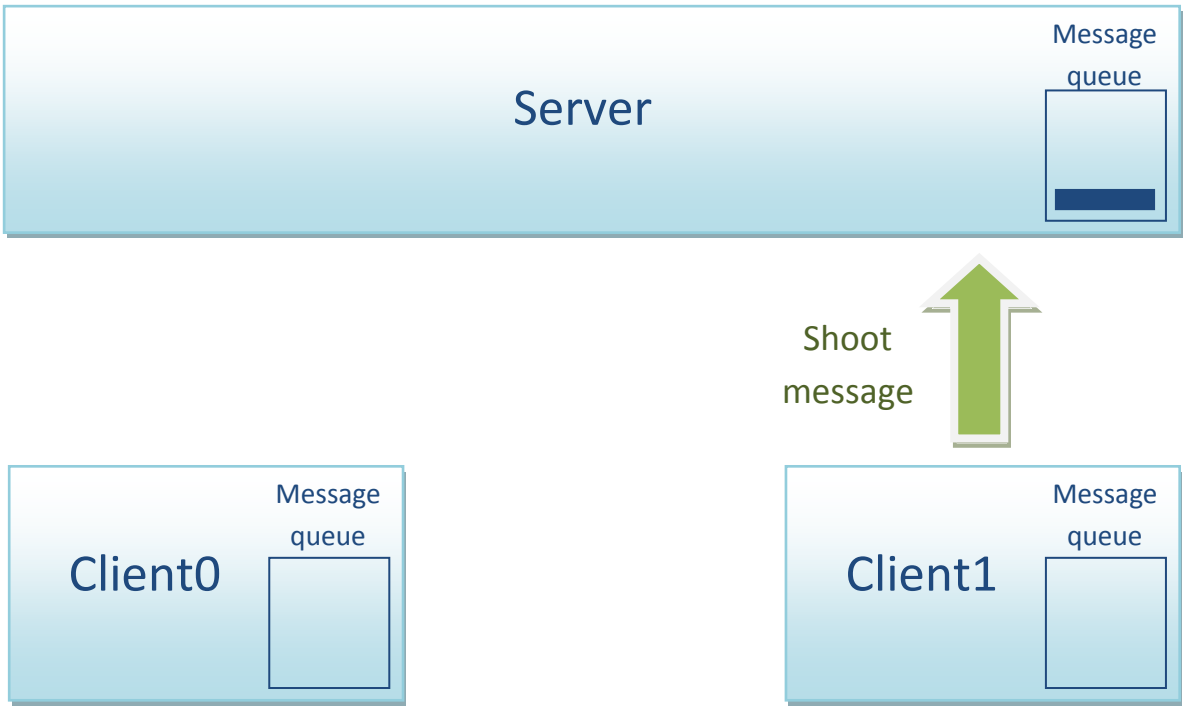

Figure 8-1. Distributed Battleship Game with no Queue Overflow

A message passing model with no queue overflow. The client sends only the shoot message to begin with.

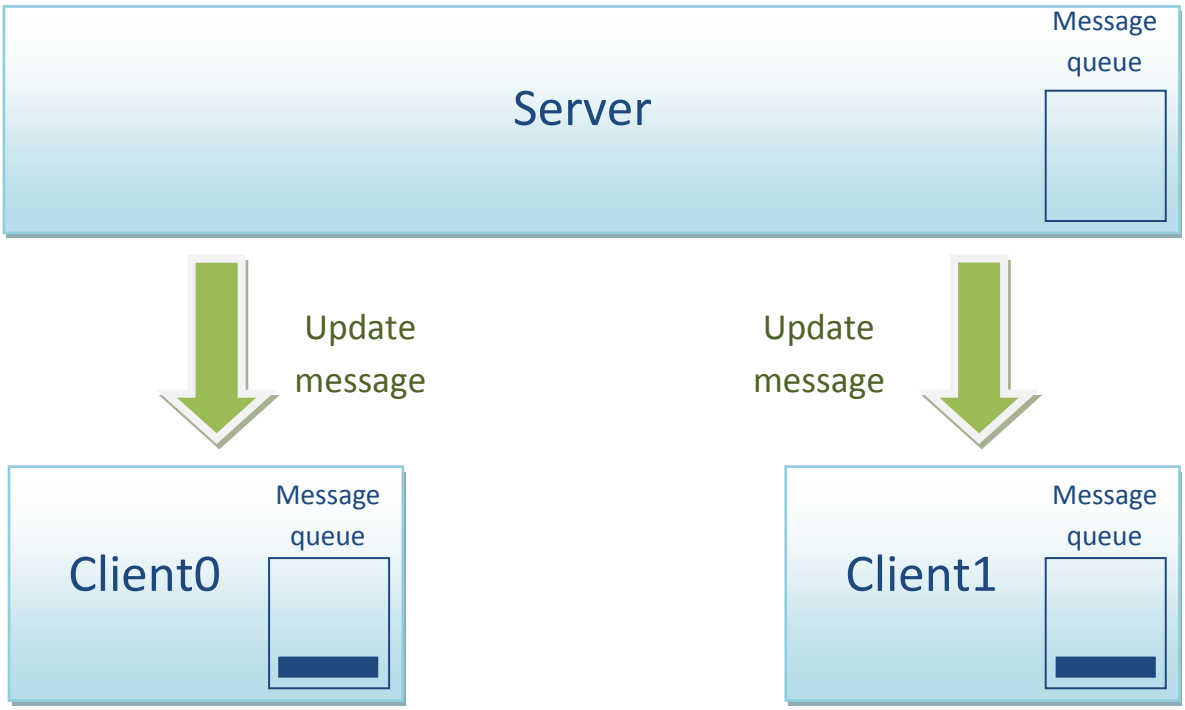

Figure 8-2. Distributed Battleship Game with no Queue Overflow

After receiving the shoot message the server multicasts the updated gamestate to the clients.

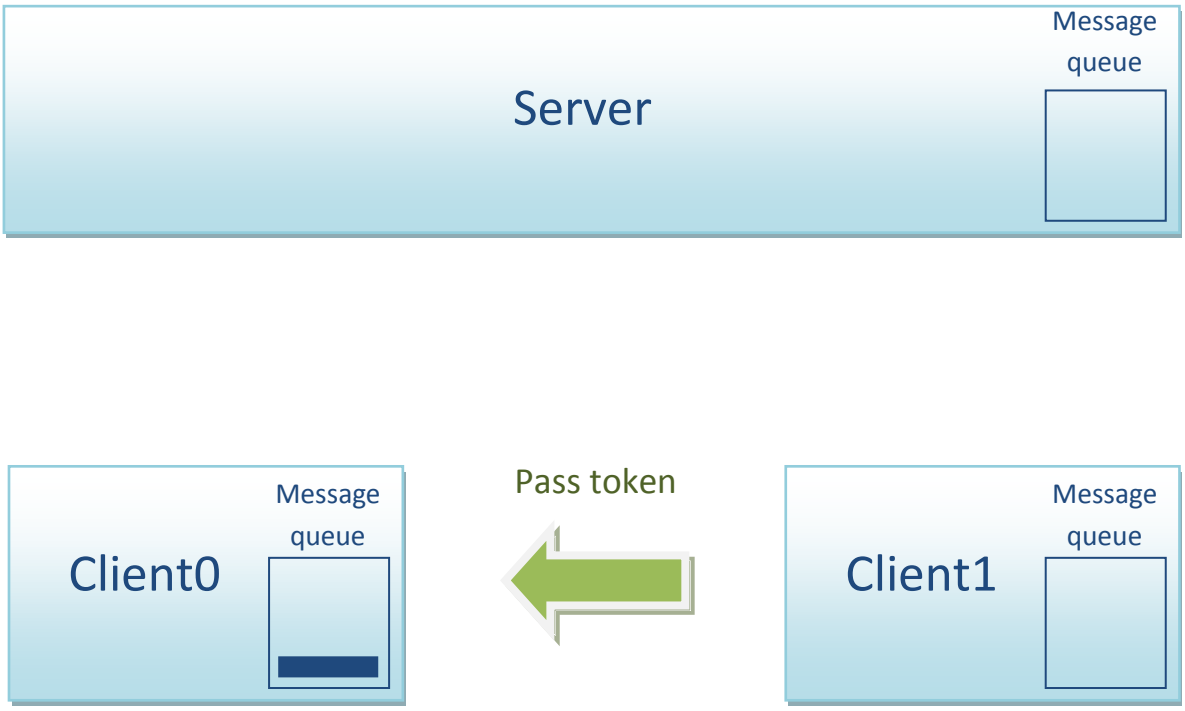

Figure 8-3. Distributed Battleship Game with no Queue Overflow

Only after the client receives the update message from the server he passes the token along.

The changes we need to apply to the model to achieve this are adding a boolean variable to the clients that tells whether the client is holding the token or not. Once he receives the token he sets the variable to true, sends a shoot message to the server and each time he receives an update message he checks whether he is holding the token. If so, he sets the variable to false and sends a token message to the next client in the token ring, resolved with a simple if-statement in the client's Shoot message server.

Code Example 2: Fixed Server and Two Clients

```
reactiveclass Server(5)
{
        knownrebecs
        {
                Client client0;
                Client client1;
        }
        statevars
        {
                int nShipsLeft;
                boolean bClient0Connected;
                boolean bClient1Connected;
        }
        msgsrv initial()
        {
                nShipsLeft = 5;
```

```
                bClient0Connected = false;
                bClient1Connected = false;
                self.NewGameAvailable();
        }
        msgsrv Join()
        {
                if( sender == client0 )
                {
                        bClient0Connected = true;
                        if( bClient1Connected == true ) client1.JoinedMessage();
                }
                else if( sender == client1 )
                {
                        bClient1Connected = true;
                        if( bClient0Connected == true ) client0.JoinedMessage();
                }
                if( bClient0Connected == true && bClient1Connected == true )
                {
                        client0.Update();
                        client1.Update();
                        client0.HasToken();
                }
        }
        msgsrv ServerShoot()
        {
                nShipsLeft = nShipsLeft - 1;
                if( nShipsLeft == 0 )
                {
                        client0.GameOver();
                        client1.GameOver();
                        self.initial();
                }
                else
                {
                        client0.Update();
                        client1.Update();
                }
        }
        msgsrv NewGameAvailable()
        {
                client0.Connect();
                client1.Connect();
        }
}

reactiveclass Client(5)
{
        knownrebecs
        {
                Server server;
                Client clientNext;
        }
        statevars
        {
                boolean bHasToken;
        }
        msgsrv initial()
        {
                bHasToken = false;
        }
        msgsrv Connect()
        {
                server.Join();
        }
        msgsrv Update()
        {
                if( bHasToken == true )
                {
                        bHasToken = false;
                        clientNext.HasToken();
                }
        }
        msgsrv HasToken()
        {
                bHasToken = true;
                server.ServerShoot();
```

```
        }
        msgsrv GameOver()
        {
                self.initial();
        }
        msgsrv JoinedMessage()
        {
        }
}

main
{
        Server server( client0, client1 ):();
        Client client0( server, client1 ):();
        Client client1( server, client0 ):();
}
```

3.2.2 Failover

The key technique for handling failures is redundancy [2]. To increase the reliability of the system we can opt to add a failover mechanism in case the server‘s operation halts for any reason.

For this to work we do have a few options:

What needs to happen is the failover knowing if the server is operational or not. Referred to as a „heartbeat“, there is the possibility of sending pulses between the server and failover in order to make this work. There is also the option of having the clients figure out whether the server is operational or not and have them take care of the problem themselves. We do however, introduce all sorts of overhead messages and duplicate information in the system that way. A better way is to have the failover take care of those elements by itself and leave the clients unchanged regarding heartbeats.

For a game to continue from the state it was in at the time of the server crash, the failover has to keep a copy of that exact state. For that to happen the failover has to continually keep track of the current gamestate so the heartbeat can be implemented as the update message needed to keep the server and failover in synchrony.

For the failover to stay updated and to figure out the operational state of the server we have two options [2]:

1. We can use a push-based approach, where updates are propagated to other replicas without those replicas even asking for the updates as in Fig-9.

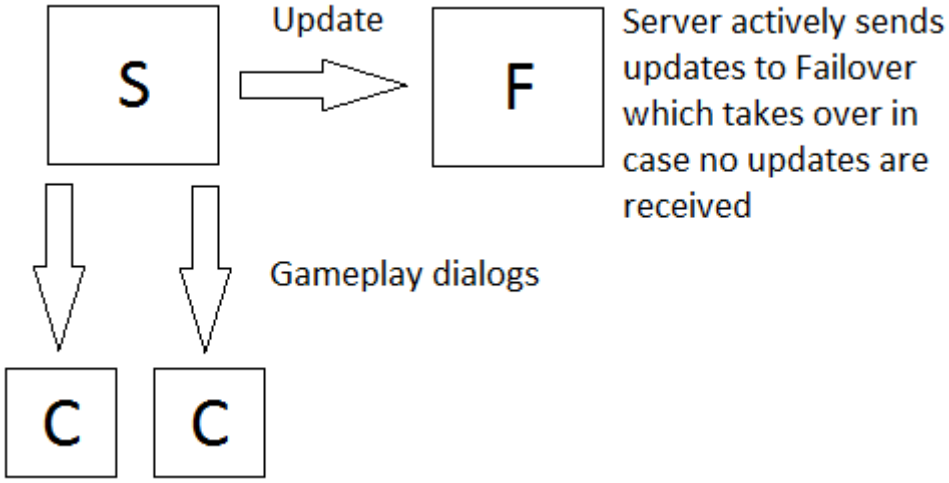

Figure 9. Push-based Failover

2. We can also go for a pull-based approach where a server or client requests another server to send it any updates it has at that moment as in Fig-10.

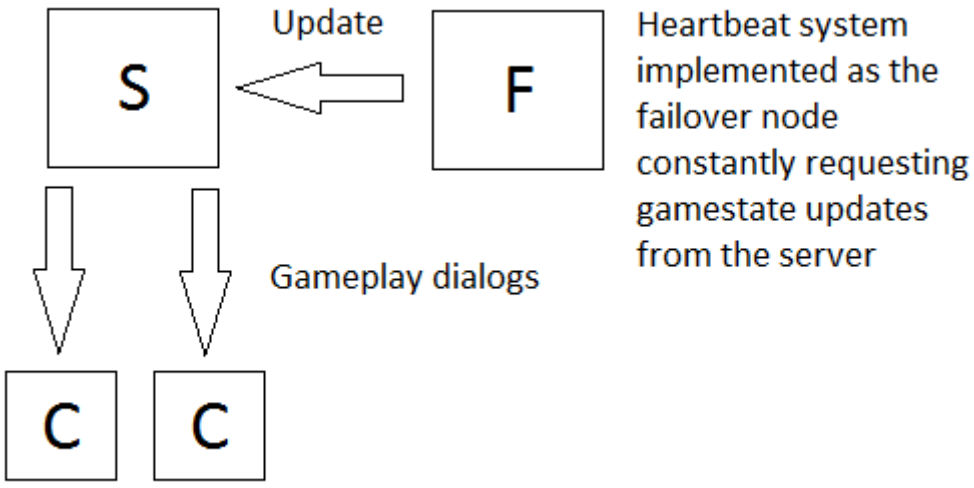

Figure 10. Pull-based Failover

Comparing these alternatives we can see that if using a push-based approach we need the server to contain information on any failover connected to it which increases coupling and lowers scalability. The server has to know about the failover and the failover

has to know about the server. The failover would need to connect to the server and disconnect if needed. If the failover goes down the server also needs a mechanism to handle that situation.

In the pull-based approach on the other hand, the failover only needs to request the updates and the server only needs to respond. This possibility therefore has lower coupling between the two and also scales better as adding more failovers requires no changes the server.

In both cases the failover still needs a timer waiting for responses to know if the server is responding or not. This can be solved by either a normal timestamp in the messages or a logical one as in clock synchronization algorithms [2].

Using this approach, in case the server does go down, the failover simply sends a message to the clients and they switch over to the failover.

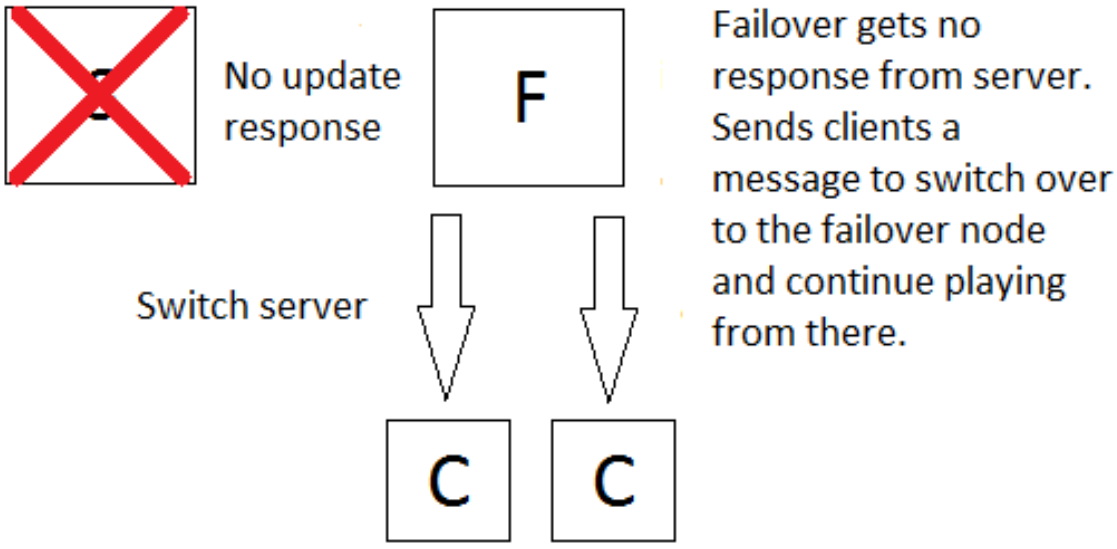

Figure 11. Failover takes over

The following code models the server going down and failover taking over after five heartbeats or update messages. We have added a new rebec named Failover which is basically a copy of the server with a few changes: it has a boolean variable for whether the

server is down or not, a heartbeat counter to simulate the server not responding at some predetermined time, and an update function for handling the communication with the server and taking over in case it's not responding.

The only changes required to the server is an update function to communicate the gamestate to the failover.

A few changes are needed to the clients. A boolean variable for whether the server is down and if he is down they communicate with the failover node instead. They also need a message server to switch between the server and failover in case the server goes down.

Code Example 3: Server, Two Clients and Failover

```
reactiveclass Failover(5)
{
	knownrebecs
	{
		Client client0;
		Client client1;
		Server server;
	}
	statevars
	{
		int nShipsLeft;
		boolean bClient0Ready;
		boolean bClient1Ready;
		int nHeartBeatCounter;
		boolean bServerDown;
	}
	msgsrv initial()
	{
		nShipsLeft = 5;
		bClient0Ready = false;
		bClient1Ready = false;
		nHeartBeatCounter = 5;
		bServerDown = false;
		server.Update();
	}
	msgsrv Update( int shipsleft, boolean client0ready, boolean client1ready )
	{
		nShipsLeft = shipsleft;
		bClient0Ready = client0ready;
		bClient1Ready = client1ready;

		if( bServerDown == false )
		{
			if( nHeartBeatCounter == 0 ) bServerDown = true;
			else
			{
				nHeartBeatCounter = nHeartBeatCounter - 1;
				server.Update();
			}
		}
		else
		{
			client0.SwitchServer();
			client1.SwitchServer();
		}
	}
	msgsrv Ready()
	{
```

```
                if( sender == client0 )
                {
                        bClient0Ready = true;
                        if( bClient1Ready == true ) client1.JoinedMessage();
                }
                else if( sender == client1 )
                {
                        bClient1Ready = true;
                        if( bClient0Ready == true ) client0.JoinedMessage();
                }
                if( bClient0Ready == true && bClient1Ready == true )
                {
                        client0.Update();
                        client1.Update();
                        client1.HasToken();
                }
        }
        msgsrv Shoot()
        {
                nShipsLeft = nShipsLeft - 1;
                if( nShipsLeft == 0 )
                {
                        client0.GameOver();
                        client1.GameOver();
                        bClient0Ready = false;
                        bClient1Ready = false;
                        nShipsLeft = 5;
                        self.NewGameAvailable();
                }
                else
                {
                        client0.Update();
                        client1.Update();
                }
        }
        msgsrv NewGameAvailable()
        {
                client0.Join();
                client1.Join();
        }
}

reactiveclass Server(5)
{
        knownrebecs
        {
                Client client0;
                Client client1;
                Failover failover;
        }
        statevars
        {
                int nShipsLeft;
                boolean bClient0Ready;
                boolean bClient1Ready;
        }
        msgsrv initial()
        {
                nShipsLeft = 5;
                bClient0Ready = false;
                bClient1Ready = false;
                self.NewGameAvailable();
        }
        msgsrv Ready()
        {
                if( sender == client0 )
                {
                        bClient0Ready = true;
                        if( bClient1Ready == true ) client1.JoinedMessage();
                }
                else if( sender == client1 )
                {
                        bClient1Ready = true;
                        if( bClient0Ready == true ) client0.JoinedMessage();
                }
                if( bClient0Ready == true && bClient1Ready == true )
```

```
                {
                        client0.Update();
                        client1.Update();
                        client0.HasToken();
                }
        }
        msgsrv Shoot()
        {
                nShipsLeft = nShipsLeft - 1;
                if( nShipsLeft == 0 )
                {
                        client0.GameOver();
                        client1.GameOver();
                        self.initial();
                }
                else
                {
                        client0.Update();
                        client1.Update();
                }
        }
        msgsrv NewGameAvailable()
        {
                client0.Join();
                client1.Join();
        }
        msgsrv Update()
        {
                failover.Update( nShipsLeft, bClient0Ready, bClient1Ready );
        }
}

reactiveclass Client(5)
{
        knownrebecs
        {
                Server server;
                Client clientNext;
                Failover failover;
        }
        statevars
        {
                boolean bHasToken;
                boolean bServerDown;
        }
        msgsrv initial()
        {
                bServerDown = false;
        }
        msgsrv Shoot()
        {
                if( bServerDown == false ) server.Shoot();
                else failover.Shoot();
        }
        msgsrv HasToken()
        {
                bHasToken = true;
                self.Shoot();
        }
        msgsrv PassToken()
        {
                bHasToken = false;
                clientNext.HasToken();
        }
        msgsrv GameOver()
        {
                bHasToken = false;
        }
        msgsrv Join()
        {
                if( bServerDown == false ) server.Ready();
                else failover.Ready();
        }
        msgsrv SwitchServer()
        {
                bServerDown = true;
```

```
        }
        msgsrv Update()
        {
                if( bHasToken == true ) self.PassToken();
        }
        msgsrv JoinedMessage()
        {
        }
}

main
{
        Server server( client0, client1, failover ):();
        Failover failover( client0, client1, server ):();
        Client client0( server, client1, failover ):();
        Client client1( server, client0, failover ):();
}
```

Bugs Found

No serious bugs found.

3.2.3 Late Join

Late join can be considered both a feature and as fault tolerance. A player might want to simply join a game in progress or a player might drop from the game and be forced to reconnect in order to continue playing. In this example we have the failover node take care of late joins.

At this point the logic within the system, such as how the servers and clients should behave whether there is a late join or not, starts to get more complex. Example of which is the Shoot message function on the server side. Because the failover handles the late joins, when the server gets a message from the failover stating there is a late join, the server needs to wait for the right moment to add the late join to the token ring. For this it needs a boolean state variable and also the normal boolean whether he‘s ready to play or not. The server also needs to know which player is the last one in the token ring in order to have that player pass his token to the latejoin instead of the first player in the token ring.

The failover is again a copy of the server with slight changes. It requires a message server to handle the late join connecting to it.

A limitation of the Rebeca model checker is that after a model starts running, it‘s not possible to change a rebecs known rebecs so in order to get a late join functionality operating we need to hardcode each rebec. That means making a different rebec for client0,

client1 and the latejoin so client1 can pass his token to the late join instead of client0 once the latejoin has joined.

Once the game starts the latejoin sends a message to the failover, the failover sends a message to the server, the server waits for client1‘s turn and then sends a message to client1 to pass his token to latejoin instead of client0. The game then continues from there and the cycle continues.

Code Example 4: Server, Two Clients, Failover and Latejoin

```
reactiveclass Server(5)
{
	knownrebecs
	{
		Failover failover;
		Client0 client0;
		Client1 client1;
		Latejoin latejoin;
	}
	statevars
	{
		int nShipsLeft;
		boolean bClient0Ready;
		boolean bClient1Ready;
		boolean bLatejoinReady;
		boolean bLatejoinJoined;
	}
	msgsrv initial()
	{
		nShipsLeft = 5;
		bClient0Ready = false;
		bClient1Ready = false;
		bLatejoinReady = false;
		bLatejoinJoined = false;
		self.NewGameAvailable();
	}
	msgsrv Ready()
	{
		if( sender == client0 )
		{
			bClient0Ready = true;
			if( bClient1Ready == true ) client1.JoinedMessage();
		}
		else if( sender == client1 )
		{
			bClient1Ready = true;
			if( bClient0Ready == true ) client0.JoinedMessage();
		}

		if( bClient0Ready == true && bClient1Ready == true )
		{
			client0.Update();
			client1.Update();
			client0.HasToken();
		}
	}
	msgsrv Shoot()
	{
		nShipsLeft = nShipsLeft - 1;
		if( sender == client1 && bLatejoinJoined == true && bLatejoinReady == false )
		{
			bLatejoinReady = true;
			client1.LatejoinJoined();
		}
```

```
                if( bLatejoinReady == false )
                {
                        if( nShipsLeft == 0 )
                        {
                                client0.GameOver();
                                client1.GameOver();
                                self.initial();
                        }
                        else
                        {
                                client0.Update();
                                client1.Update();
                        }
                }
                else
                {
                        if( nShipsLeft == 0 )
                        {
                                client0.GameOver();
                                client1.GameOver();
                                latejoin.GameOver();
                                self.initial();
                        }
                        else
                        {
                                client0.Update();
                                client1.Update();
                                latejoin.Update();
                        }
                }
        }
        msgsrv NewGameAvailable()
        {
                client0.Join();
                client1.Join();
        }
        msgsrv Update()
        {
                failover.Update( nShipsLeft, bClient0Ready, bClient1Ready, bLatejoinJoined,
bLatejoinReady );
        }
        msgsrv LatejoinServer()
        {
                bLatejoinJoined = true;
        }
}

reactiveclass Failover(5)
{
        knownrebecs
        {
                Server server;
                Latejoin latejoin;
        }
        statevars
        {
                int nShipsLeft;
                boolean bClient0Ready;
                boolean bClient1Ready;
                boolean bLatejoinJoined;
                boolean bLatejoinReady;
        }
        msgsrv initial()
        {
                nShipsLeft = 5;
                bClient0Ready = false;
                bClient1Ready = false;
                bLatejoinJoined = false;
                bLatejoinReady = false;
                server.Update();
        }
        msgsrv Update( int shipsleft, boolean client0ready, boolean client1ready, boolean
latejoinJoined, boolean latejoinReady )
        {
                nShipsLeft = shipsleft;
                bClient0Ready = client0ready;
```

```
                bClient1Ready = client1ready;
                bLatejoinJoined = latejoinJoined;
                bLatejoinReady = latejoinReady;
        }
        msgsrv LatejoinFailover()
        {
                server.LatejoinServer();
        }
}

reactiveclass Client0(5)
{
        knownrebecs
        {
                Server server;
                Client1 client1;
        }
        statevars
        {
                boolean bHasToken;
        }
        msgsrv initial()
        {
                bHasToken = false;
        }
        msgsrv Shoot()
        {
                server.Shoot();
        }
        msgsrv HasToken()
        {
                bHasToken = true;
                self.Shoot();
        }
        msgsrv PassToken()
        {
                bHasToken = false;
                client1.HasToken();
        }
        msgsrv GameOver()
        {
                self.initial();
        }
        msgsrv Join()
        {
                server.Ready();
        }
        msgsrv Update()
        {
                if( bHasToken == true ) self.PassToken();
        }
        msgsrv JoinedMessage()
        {
        }
}

reactiveclass Client1(5)
{
        knownrebecs
        {
                Server server;
                Client0 client0;
                Latejoin latejoin;
        }
        statevars
        {
                boolean bHasToken;
                boolean bLatejoinJoined;
        }
        msgsrv initial()
        {
                bHasToken = false;
                bLatejoinJoined = false;
        }
        msgsrv Shoot()
        {
```

```
                server.Shoot();
        }
        msgsrv HasToken()
        {
                bHasToken = true;
                self.Shoot();
        }
        msgsrv PassToken()
        {
                bHasToken = false;
                if( bLatejoinJoined == true ) latejoin.HasToken();
                else client0.HasToken();
        }
        msgsrv GameOver()
        {
                self.initial();
        }
        msgsrv Join()
        {
                server.Ready();
        }
        msgsrv Update()
        {
                if( bHasToken == true ) self.PassToken();
        }
        msgsrv JoinedMessage()
        {
        }
        msgsrv LatejoinJoined()
        {
                bLatejoinJoined = true;
        }
}

reactiveclass Latejoin(5)
{
        knownrebecs
        {
                Server server;
                Failover failover;
                Client0 client0;
        }
        statevars
        {
                boolean bHasToken;
        }
        msgsrv initial()
        {
                bHasToken = false;
                failover.LatejoinFailover();
        }
        msgsrv Shoot()
        {
                server.Shoot();
        }
        msgsrv HasToken()
        {
                bHasToken = true;
                self.Shoot();
        }
        msgsrv PassToken()
        {
                bHasToken = false;
                client0.HasToken();
        }
        msgsrv GameOver()
        {
                self.initial();
        }
        msgsrv Join()
        {
                server.Ready();
        }
        msgsrv Update()
        {
                if( bHasToken == true ) self.PassToken();
```

```
        }
        msgsrv Joined()
        {
        }
}

main
{
        Server server( failover, client0, client1, latejoin ):();
        Failover failover( server, latejoin ):();
        Client0 client0( server, client1 ):();
        Client1 client1( server, client0, latejoin ):();
        Latejoin latejoin( server, failover, client0 ):();
}
```

Bugs Found

No serious bugs were found while modeling the late join mechanism. The logic of the system did get more complex but working with it on this layer of the abstraction without any details getting in the way made it much easier as compared with testing through code.

3.2.4 Rebeca Experiment Results

Following are the results of the model checker for each of the models. The results are printed out below the code window and show whether a model is satisfied, contains a queue overflow or a deadlock.

If there is a problem with the model the model checker prints out the first state it checked to begin with. It shows global variables if any, the system information, states of the rebecs and its possible to browse through the states to trace the series of events that lead to the error.

The model checker uses a graph search [5] so the first variable for the system information is the how deeply it reached into the graph before an error occurred. It also shows the maximum reached state which means how many states in the graph were checked. It is possible to look for errors using different sets of rules. In our case we used Linear Temporal Logic (LTL) as the system‘s default deadlock. The result is then shown as either deadlock, queue overflow or satisfied.

Following the system information comes information regarding the states of the rebecs during each state of the system in case an error was detected. It lists the states of their variables, the messages in their queues and who sent them in their respective order.

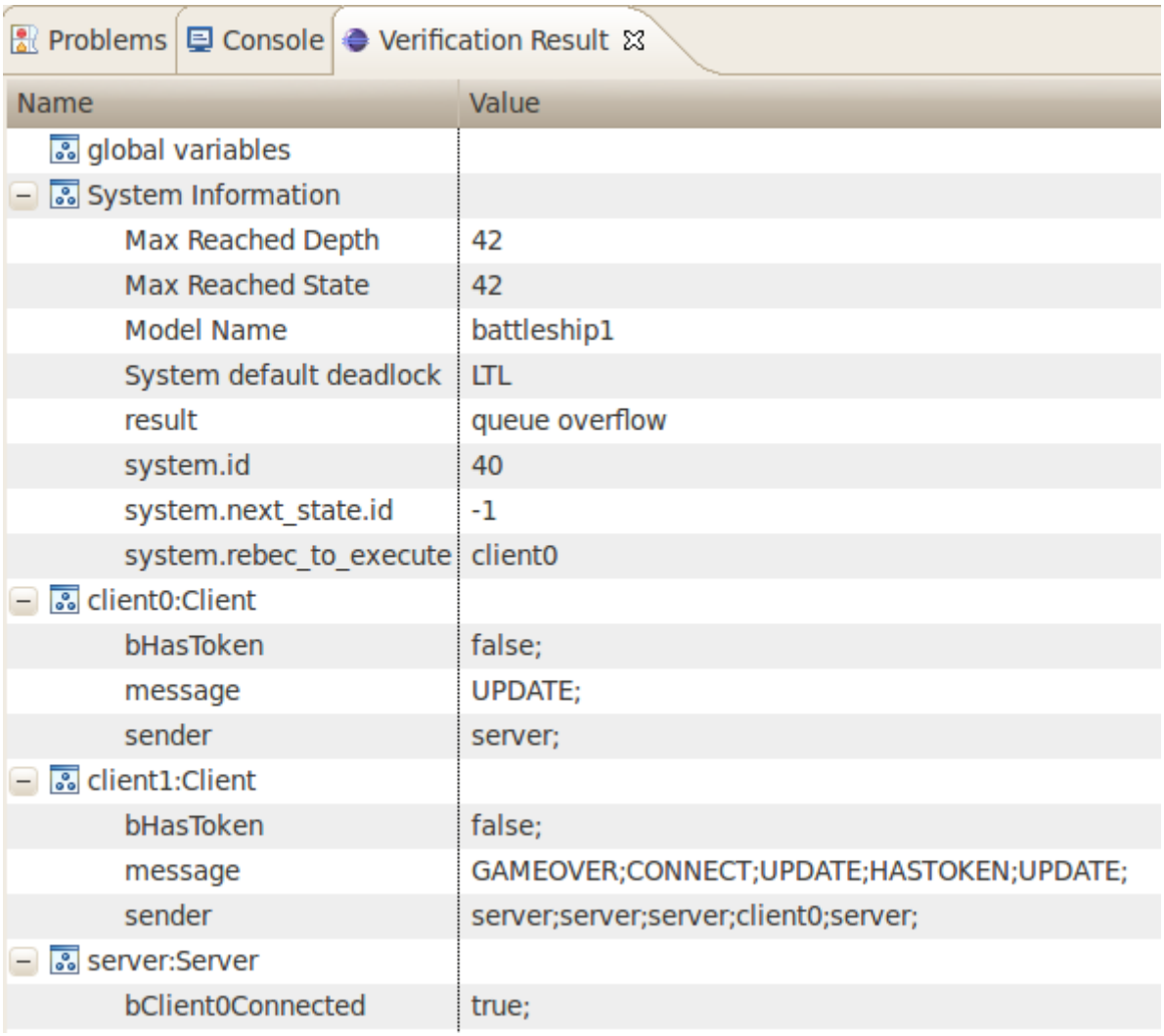

Figure 12. Results of Queue Overflow

A queue overflow with a server and two clients as output by Rebeca. Shows the system as the state of the state variables and message queues. In this case, client1's message queue has overflowed due to too many messages being sent to it each turn. The clients are sending shoot and token messages simultaneously.

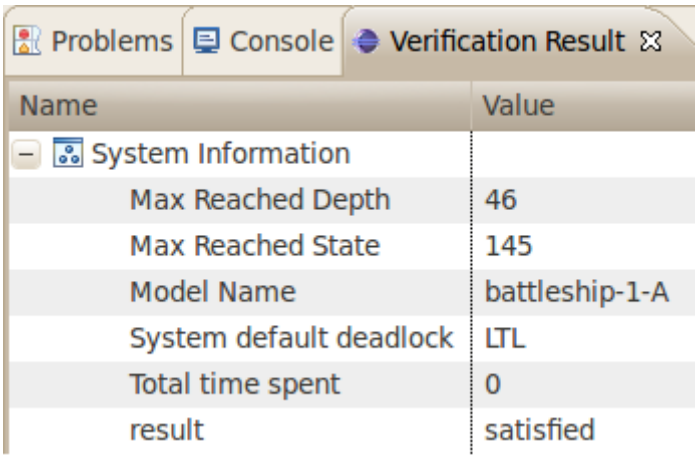

Figure 13. Server and two Clients satisfied

Verification results of the same system but this time the clients wait for update messages from the server before passing tokens.

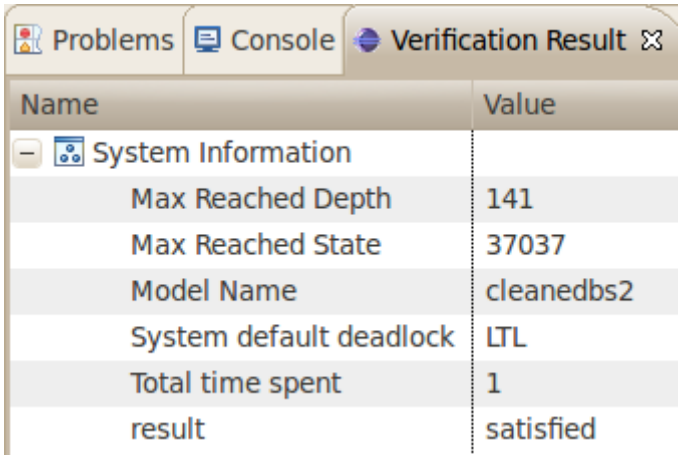

Figure 14. Server, two clients and Failover satisfied

Verification of a server, two clients and a failover node. Note the size of the state space (Max Reached State) as compared with the previous model.

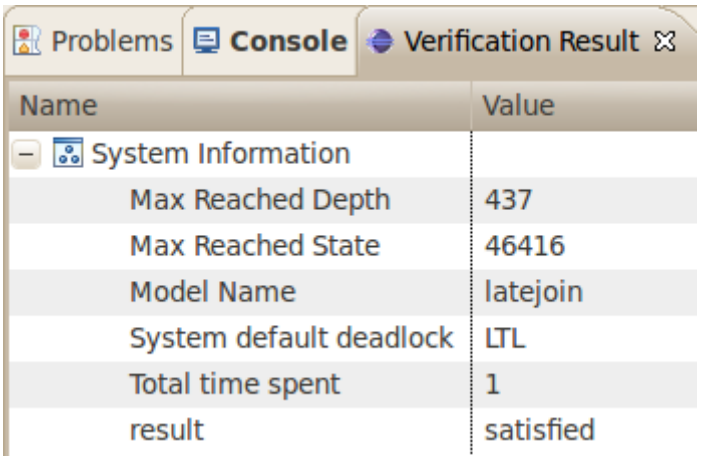

Figure 15. Server, two clients, Failover and Latejoin satisfied

Latejoin verification shows a larger state space and greatly increased depth. Foregoing a heartbeatcounter because we're only testing the latejoin part greatly reduces the state space but more boolean state variables are needed to accomodate the latejoin mechanism.

4. Discussion

By now it should be clear that both because of the state space explosion and the complexity of developing a distributed system, without modeling it can result in defects slipping through. What‘s missing from the report is that while creating the models, dozens, even hundreds of deadlocks and queue overflow‘s were detected by the model checker not because of design issues but because of programmer error. These were not documented as it would simply have been too much. The alternative would have been to directly do the coding and run a test which would have taken a lot more time and effort. Also, due to the complexity of that kind of test we might have introduced other errors that might not have had anything to do with the message passing layer. That kind of development process inherently uses up a lot more time as if there is an error, we might not necessarily know on which layer of the abstraction it is taking place. For example, it could be on the network layer, transport layer, application layer, message passing layer or simply a typo somewhere. By using a model checker we can start eliminating possibilities right away and view the system from a standpoint where it‘s easy to see the forest for the trees.

4.1 Patterns

A pattern noticed while designing and modeling was that whenever there were alternative design choices to be made, the decision usually came down to one of three factors: number of messages required, coupling between modules and scalability. As more messages are being passed around not only does the system become error-prone but the complexity increases. With increased coupling the system tended to both scale worse and more messages were required. Controlling the number of messages and the coupling between the modules tended to scale better and if it scaled well the number of messages was usually low and the modules less coupled. Having these factors constantly in check controlled the complexity and hence both its efficiency and ease of maintenance. Different systems with different requirements might operate differently however.

Another pattern noticed was incomplete if-else statements. This could result in messages being sent that were irrelevant to the current state of the system and therefore do nothing except take up space in message queues. Although not introducing much risk by

themselves, if a few of these different messages slip through, in which case they would go unnoticed during testing because they don‘t effect the current state, they can start effecting the systems realiability and efficiency once enough of them are being passed around. The model checker fortunately catches these kind of things for us.

Perhaps the most important pattern though was the passing of multiple messages simultaneously as in the case of the shooting and token passing example. By breaking down the communication and having the shooter wait for an update message, as a sort of acknowledgement message, the problem was resolved. If the system had both these types of design flaws and other things such as incomplete if-elses, we can see that right away the system starts becoming less and less reliable. As an example from probability theory: If the probability of one error happening is two percent, and the probability of some other error is three percent, the probability of either one of them taking place is five percent. In our case though the difference is that both errors affect the message queues so the combined effect is unknown before it actually takes place. This type of error might however not have been discovered through testing alone.

4.2 Integrating into the Development Cycle

For integrating model checking into the development cycle there are options: it is possible to design and verify the whole system before coding, code then verify or model and code iteratively. Which way to go may depend on the situation. A system might turn out to be so modular no changes are ever required to a module once coded. In that case modeling the whole thing and then coding might seem an option. Iteratively it could be possible to model a certain mechanism and then code right away. Some might simply be so good at making distributed systems they never have a problem. In that case verifying the system after coding might seem an option. Depending on how mission-critical a system is, modelling might be skipped altogether, but as we have seen, it might actually save more time than it takes during the development process, no matter it‘s importance.

5. Conclusion and future work

Testing can prove the existence of bugs but not that they don‘t exist. Due to the state space explosion it becomes virtually impossible to test every single state of the system. We could run into a bug somewhere during the testing process but whether all of them were found we won‘t know for sure.

Model checking can prove both the existence of bugs and that they don‘t exist. Used in conjuction with testing, results in a more correct, reliable, efficient distributed system.

As future work we can consider integrating more ack or acknowledgement messages into the model. This would require the clients to contain information about the previous node in the token ring and pass more messages around which introduces overhead but results in higher reliability if done correctly. Many of the messages cannot be lost or the system deadlocks so some way of resending messages if needed is required. This also introduces the ability for the system to know if a client has dropped or not.

Having the server capable of handling latejoins would also be a nice feature.

Appendix A

Coffman conditions for deadlock

For a deadlock to occur there are four necessary conditions [3]:

1. Mutual exclusion condition: a resource that cannot be used by more than one process at a time.

2. Hold and wait condition: processes already holding resources may request new resources.

3. No preemption condition: No resource can be forcibly removed from a process holding it, resources can be released only by the explicit action of the process.

4. Circular wait condition: two or more processes form a circular chain where each process waits for a resource that the next process in the chain holds.

Appendix B

Threads

Examples of thread uses which might and might not be solved with message passing: *(Non-exhaustive.)*

1. *„Some applications can very effectively use threads. So-called „embarrassingly parallel“ applications (for example, applications that essentially spawn multiple independent processes such as build tools, like PVM gmake, or web servers). Because of the independence of these applications, programming is relatively easy, and the abstraction being used is more like processes that threads (where memory is not shared). Where such applications do share data, they do so through database abstractions, which manage concurrency through such mechanisms as transactions. However, client-side applications are not so simple [4].“*

2. Maintaining a responsive user interface: threads can run time-consuming tasks while main UI threads process keyboard and mouse events.

3. Making efficient use of an otherwise blocked CPU: if a thread is blocked or waiting for a response from another process or thread, other threads can use the CPU in the meantime.

4. Parallel programming: „divide-and-conquer“ algorithms can split up their workloads to make better use of multiple processors.

5. Speculative execution: if we don‘t know which algorithm is optimal for a certain job, we can execute them in parallel and whoever finishes first wins.

6. Allowing requests to be processed simultanously: a server can receive client requests concurrently and process them in parallel. Can also be useful for clients in a peer-to-peer network or even handling requests from a user.

Non-exhaustive list of common problems with shared memory

1. Introduces bugs that are hard to replicate.
2. The state space grows exponentially.
3. Programs that crash in a critical region.
4. Programs that spend too much time in the critical region.
5. Locking too much – can be optimized to use smaller regions of memory.
6. Distributed shared memory – too complex, single fault tolerant databases used instead.
7. Sharing limits scalability – doesn‘t matter on a single cpu but on a multicore locking causes the system to operate serially which defeats the whole purpose of multiple cpus and parallelism.
8. Sharing makes systems error prone and debugging difficult – process A can overwrite data from B, problem not in the code for B.

Items without reference are based on different articles, forum comments, and miscallenous Internet searches.

Terminology

Asynchronous vs. synchronous:

„Besides being persistent or transient, communication can also be asynchronous or synchronous. The characteristic feature of asynchronous communication, is that a sender continues immediately after it has submitted its message for transmission. This means that the message is stored in a local buffer at the sending host, or otherwise at the first communication server. With synchronous communication, the sender is blocked until its message is stored in a local buffer at the receiving host, or actually delivered to the receiver. The strongest form of synchronous communication is when the sender is blocked until the receiver has processed the message [2].“ Synchronous communication uses locks or blocking while asynchronous doesn‘t. Asynchronous systems rely on buffers or queues while synchronous don‘t.

„Hiding communication latencies is applicable in the case of geographical scalability. The basic idea is simple: try to avoid waiting for responses to remote service requests as much as possible. For example, when a service has been requested at a remote machine, an alternative to waiting for a reply from the server is to do other useful work at the requester‘s side. Essentially, this means constructing the requesting application in such a way that it uses only asynchronous communication. Asynchronous communication can often be used in batch-processing systems and parallel applications, in which more or less independent tasks can be scheduled for execution while another task is waiting for communication to complete. Alternatively, a new thread of control can be started to perform the request. Although it blocks waiting for the reply, other threads in the process can continue [2].“

Concurrency: A condition that exists when at least two threads are making progress. May or may not be executing in parallel. Property of systems in which several computations are executing simultaneously and potentially interacting with each other.

Coupling: or dependency is the degree to which each program module relies on each one of the other modules.

Critical section/region: The part of the program where the shared memory is accessed is called the critical region or critical section.

Distributed system: *„A distributed system is a collection of independent computers that appears to its users as a single coherent system [2].“*. Consists of multiple autonomous computers that communicate through a computer network. No shared memory.

Livelock: Two or more processes constantly change their states in regard to one another. A real world example of a livelock occurs when two people meet in a narrow corridor, and each tries to be polite by moving aside to let the other pass, but they end up swaying from side to side without making any progress.

Model: Set of rebecs as a closed system composed of rebecs, which are concurrently executed and are interacting with each other [5].

Mutual exclusion: a way of prohibiting more than one process from reading and writing the shared data at the same time. Also referred to as „blocking“.

Mutex: *„A variable that can be in one of two states: unlocked or locked [2].“* Sometimes referred to as the „lock“.

Parallel system: A system with at least two processes that use shared memory. Usually refers to multicore systems or systems with multiple processors.

Parallelism: A condition that arises when at least two threads are executing simultaneously or in parallel.

Process: *„A process is basically a program in execution [1].“* Instance of a computer program that is being executed. Can include multiple threads.

Reactive object (rebec): Object within a reactive system that has interactions with other objects [5].

Reactive systems: *„Systems which have ongoing interactions with their environments, accepting requests and producing responses [5].“ „Computes by reacting to stimuli from its environment [6].“*

Register: Processor register. Small amount of storage available on the CPU whose contents can be accessed more quickly than storage available elsewhere.

Scalability: A desirable property of a system, a network or a process, which indicates its ability to either handle growing amounts of work in a graceful manner or to be enlarged.

Shared memory: memory that may be simultaneously accessed by multiple threads.

Starvation: a process is perpetually denied access to a resource.

State space explosion: the effect that adding variables to a closed system increases the number of possible states within that system exponentially. Example: system with one boolean variable has two possible states, true or false. Add another boolean and the state space is now two times two or four possible states within the system. Each addition of a variable multiplies the current state space by the number of possible values the new variable can hold.

Thread: *„In traditional operating systems, each process has an address space and a single thread of control. In fact, that is almost the definition of a process. Nevertheless, there are frequently situations in which it is desirable to have multiple threads of control in the same address space running in quasi-parallel, as though they were (almost) separate processes (except for the shared address space) [1].“* Smallest unit of processing that can be scheduled by an operating system. Process with multiple threads is referred to as being multithreaded.

References

[1] Tanenbaum, A. S.: *Modern Operating Systems (3rd edition)*, Prentice Hall, Upper Saddle River, NJ, USA, 2008.

[2] Tanenbaum, A. S. van Steen, M.: *Distributed Systems: Principles and Paradigms*, Upper Saddle River, NJ, USA, 2002.

[3] Coffman, E.G., Elphick, M. J., Shoshani, A.: *System Deadlocks*, ACM Computing Surveys Volume 3 Issue 2, June 1971.

[4] Lee, E. A.: *The Problem with Threads*, Electrical Engineering and Computer Sciences University of California at Berkeley, Technical Report No. UCB/EECS-2006-1, USA, January 2006.

[5] Sirjani, M., Movaghar, A., Shali, A., de Boer, F. S.: *Modeling and Verification of Reactive Systems using Rebeca*, IOS Press, Fundamenta Informaticae 63 1-26, Netherlands, 2004.

[6] Aceto, L., Larsen K. G., Ingólfsdóttir A.: *Reactive Systems: Modelling, Specification and Verification*, BRICS, Department of Computer Science, Aalborg University, Denmark, May 11, 2005.

[7] Manna, Z., Pnueli, A.: *The Temporal logic of Reactive and Concurrent Systems*, Springer-Verlag, Berlin, Germany, 1992.

[8] Cheriton, D. Skeen, D.: *Understanding the Limitations of Causally and Totally Ordered Communication*, Proc. 14th Symp. Operating System Principles, ACM, USA, 1993.

[9] Olofsson, P.: *Probability, Statistics, and Stochastic Processes*, Wiley-Interscience, UK, 2002.