View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by OAR@UM

Distributed Protocols as Behaviours in Erlang

Darren Demicoli
CS, ICT
Univeristy of Malta
ddem0003@um.edu.mt

Adrian Francalanza
CS, ICT
Univeristy of Malta
adrian.francalanza@um.edu.mt

ABSTRACT

We investigate the implementation of standard algorithms for three classes of Distributed Agreement problems in Erlang, an industry-strength language for programming fault-tolerant distributed systems. We develop a framework to bridge the gap between the assumptions of these standard algorithm and the network abstraction provided by Erlang, and structure our implementations as reusable behaviours within this framework.

Keywords

Distributed Algorithms, Erlang, Fault-Tolerance

1. INTRODUCTION

Programming reliable distributed systems is hard due to the additional complexities introduced by the decentralised underlying architecture, such as multiple interleavings of distributed code, network asynchrony, *i.e.,* no guarantees on execution and communication times, and node and link failures. These aspects increase the potential for errors in the code and make it hard to reproduce (and hence debug) errors.

Various literature exists on how to program distributed systems reliably[7, 6]. Most of these argue that distributed programs can often be distilled into, and expressed as, a collection of recurring distributed problems such as Atomic Commit and Reliable Broadcast. Devising algorithms for these problems is an active area of research and a number of established algorithms exist that are know to correctly implement a solution to these distributed problems.

In this paper we report on the implementation of a number of these algorithms in Erlang [1], an industry-strength functional language for programming fault-tolerant distributed systems. We package these algorithm implementations as modules offering suite of protocols (Erlang behaviours) which follow the host language's code practices such as those dictated by the Erlang/OTP standards[8]. This structure of code organisation is attractive for a number of reasons: it abstracts the complexities of these algorithms underneath the protocol interface, it standardises code and it also simplifies the development and maintainability of Erlang distributed applications through code reuse.

Although most of these algorithms are well-understood and verified correct, their implementation poses problems even in a domain specific language such as Erlang. Problems arise due to *semantic incompatibilities i.e.,* when the algorithms assume a model that has no direct mapping in the host language and *context adaptations i.e.,* additions and modifications to the algorithm to satisfy additional requirements imposed by the host language[10]. The paper discusses such obstacles encountered and describes how we addressed them in our implementation.

The paper is structured as follows. Sec. 2 introduces the running example to be implemented in Erlang and Sec. 3 outlines the main obstacles encountered. Sec. 4 and Sec. 5 discuss our solution to these problems and Sec. 6 evaluates our solution. Sec. 7 concludes with directions for future work.

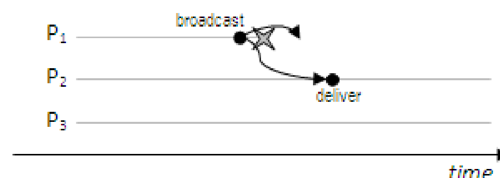

Figure 1: Node failure with an unreliable broadcast algorithm

2. MOTIVATION

Consider a distributed system in which participant nodes have the following operations:

- *File Transfer*: sending files to all other nodes, which is then stored by the receivers.
- *File Execution*: executing files at all remote nodes.

In order to handle the complexities associated with distributed systems, both operations can be expressed in terms of *Reliable Broadcast*, a distributed problem specifying functionality for reliably sending messages to all nodes (*broadcast*) and *receiving* messages broadcast by other nodes. Implementing an algorithm that solves Reliable Broadcast is non-trivial. Fig. 1 outlines what may happen when the sender crashes whilst broadcasting and only manages to send the message to just P_2 before failing, leaving the system in an inconsistent state since P_3 never gets the message.

Given the *complexity* of the algorithm required to solve this problem (*cf.* Fig. 3) and its *reuse* in the implementation of the two operations, it makes sense to abstract its implementation and package it as a reusable unit which can be called through some specified interface, as shown in Fig. 2. This code organisation carries obvious advantages because

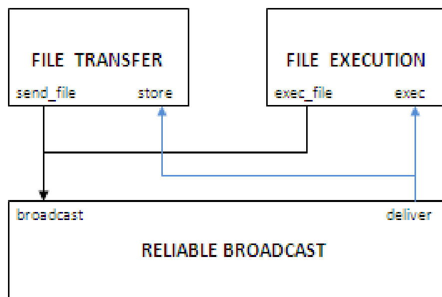

Figure 2: Usage of a Reliable Broadcast module for reliable file transfer and execution

it enables testing of the algorithm in isolation, standardises implementation, reduces the risk of errors in the code and expedites development.

In fact, Reliable Broadcast embodies just one class of distributed algorithms known as Agreement Problems[7][6]; other classes include *Distributed Consensus*, where every participants agree on some decision based on a proposal given by the participants themselves and *Atomic Commit*, where an action must be carried out by all participants or none.

Descriptions of algorithms solving these problems follow an asynchronous event based model, whereby *event handlers* execute concurrently and in isolation, and may, in turn, trigger other event handler by generating *events* themselves. Moreover their presentation is often organised in terms of module interfaces consisting of:

External events handled: These are events that can be triggered by the user of this module. For example, *Regular Reliable Broadcast*, handles the *broadcast* event, to send messages to all processes.

Callbacks Expected: These are event handlers that he user of a module is expected to implement. For example, the *Regular Reliable Broadcast* callbacks the *deliver* event, which should indicate that a message was received from a process.

The host language chosen, Erlang, provides a level of abstraction that lends itself well to the implementation of algorithms described in this manner. Erlang processes are an integral part of the language and are lightweight enough to be used directly to model event handlers. Moreover, these processes interact through message-passing which can, in turn, be used to model in straightforward fashion asynchronous events. Crucially though, Erlang allows one to structure code as *behaviours*, modules that, apart from specifying the signature of functions implemented by the module, specify also signatures of functions that are expected to be implemented by the used of the module. This allows one to closely follow the algorithm specification expressed in terms of module interfaces discussed above with external event handlers and callbacks.

3. IMPLEMENTATION OBSTACLES

Despite these advantages, the implementation of these agreement algorithms in Erlang still posed problems. In order to facilitate our explanations, we first look in more detail

```
 1  module regular_reliable_broadcast
 2
 3  uses best_effort_broadcast
 4  uses perfect_failure_detector
 5
 6  %% Event Handlers
 7  upon event init() do
 8    Delivered = ∅,
 9    Correct = Π,
10    ∀ P∈Π · From[P] = ∅.
11
12  upon event broadcast(Msg) do
13   trigger best_effort_broadcast:broadcast({self(), Msg}).
14
15  upon event beb_deliver(Relay, {Sender, Msg}) do
16    if ({Sender, Msg} ∉ Delivered) then
17       Delivered = Delivered ∪ { {Sender, Msg} }
18       callback rrb_deliver( Sender, Msg )
19       From[Relay] = From[Relay] ∪ { {Sender, Msg} }
20       if Relay ∉ Correct then
21           trigger best_effort_broadcast:broadcast({Sender, Msg})
22
23  upon event crash(Who) do
24    Correct = Correct \ {Who}
25    ∀{Sender, Msg}∈ From(Who)·
26           trigger best_effort_broadcast:broadcast({Sender, Msg})
```

Figure 3: Regular Reliable Broadcast Algorithm

at the specification of Regular Reliable Broadcast from our running example of Fig. 2. This will enable us to pinpoint the issues involved and allow the reader to better appreciate our approach.

3.1 Regular Reliable Broadcast

This specification is expressed in terms of the following three properties:

Agreement: All correct participants deliver the same set of broadcast messages.

Validity: If a correct participant p broadcasts message m, this eventually gets delivered by all correct participants in the system.

Integrity: For any message m, every participant delivers m at most once, and only if it was previously broadcast.

The algorithm solving the Regular Reliable Broadcast specification requires every participant to store all the messages it delivers. Whenever its failure detector detects that some participant crashed, it will relay all the messages it has received from this crashed participant to the rest of the participants, on the premise that this crashed participant could have crashed without being able to successfully send the message to all the participants. Since when a participant crashes, multiple participants might attempt to relay this message to other participants, every participant also needs to keep a set of all the messages it delivered (*Delivered*), in order to ensure that every message only gets delivered once.

Fig. 3 presents an algorithm for the *Regular Reliable Broadcast*, adapted from [6] following the asynchronous event-based programming model discussed in Sec. 2. It contains event handlers for the *init*, *broadcast*, *beb_deliver* and *crash* events. The first two events are part of the interface of the algorithm. The *beb_deliver* event is a callback of the *best_effort_broadcast* module being used by this module (line 3). The *best effort broadcast* is a broadcast specification which just sends the message to all processes, one after the other (but provides no agreement guarantees if the sender

crashes). The *beb_deliver* callback indicates that a message was received through the *Best_effort_broadcast* module. Line 13 is an example where the *broadcast* event of the *best_effort_module* is being triggered. The *crash(Who)* event is a callback of the *perfect_failure_detector* module (line 4). This indicates that the process whose PID is Who, crashed. Note also that the algorithm uses a number of global variables: *Delivered*, *Correct* and *From*. The first two are sets, whereas *From* is a mapping between PID's and sets of messages.

The algorithm relies on an *asynchronous system model* where no assumption is made on the delays of messages exchanged and on the execution times. It also assumes a *crash-failure* model of failure: participants which fail are assumed to remain in this failed state forever. Participants which do not crash are said to be *correct*, whereas participants which do crash are referred to as *faulty*. It also relies on the concept of *failure detectors* [2], an abstraction providing information about the state of participants (whether they crashed or not). In particular it assumes a *perfect failure detector*, guaranteeing *completeness i.e.,* all failed participants are detected, and *accuracy i.e.,* all correct participants are not detected to be failed.

3.2 Implementing Regular Reliable Broadcast

Implementing the algorithm in Fig. 3 in Erlang is not as straightforward as one would expect. It poses the following problems:

Immutable State: The algorithm makes use of global variables (such as the *correct* and *delivered* sets) and assumes a shared memory programming model. However, Erlang functions have an immutable state and Erlang processes communicate exclusively through message passing and not global variables.

Asynchronous Event Functionality: Although Erlang's message passing mechanism model is analogous to the asynchronous event based model followed by the algorithm, some work is required to implement *predicate triggered events* - events which instead of being triggered by an external module, are triggered when a particular boolean predicate becomes true.

Process Communication: The algorithm in Fig. 3 communicate at a participant to participant level which requires all senders to know the PID (or a globally unique name) of the receiver. In our implementation, however, these participants are decoposed into processes (one per event handler) and communication happens at a process to process level. Moreover, these processes are distributed across nodes (one node for every participant) each of which has a globally unique name. It would be more practical for processes to send messages to a gateway process on every node which forwards the messages to all the relevant process.

Perfect Failure Detector: The agreement algorithms, considered in this work, assume a perfect failure detector. Erlang comes equipped with its own failure detector which, however, does not guarantee *strong accuracy*. In particular, the heartbeat mechanism used can erroneously suspect that a participant has died. In order to implement these algorithms, further work is required to come up with a failure detector which satisfies the *strong accuracy* properties.

Unreliable Communication: Distributed process communicate by exchanging messages. Nevertheless, messages can be lost after being sent and never reach their recipient. However, formal algorithms assume perfect communication links meaning that messages sent will eventually be delivered. Clearly, this can cause problems because messages might never reach their destination and hence a mechanism to guarantee delivery is required.

4. ERLANG IMPLEMENTATION

4.1 Implementation Framework

The agreement algorithms implemlemented in this work share a number of similarities. A common implementation framework was devised so as to (*i*) provide a common structure to these algorithms (*ii*) standardise the implementation of certain mechanisms common to all algorithms. This framework provides a coherent way by which to implement *asynchronous event based* algorithms on top Erlang's message passing model. It structures the implementation of every algorithm as a sequence of responses to particular events. During an execution of the algorithm, events are sent as messages and reside in the receiver process' mailbox. Moreover, the framwork standarises the encoding and access of the internal state of an algorithm; it is maintained an Erlang tuple [1] which is retrieved and updated by every event handler.

This framework also forces all algorithms to be implemented as Erlang behaviours. Behaviours are an Erlang construct by which strict interfaces for module composition can be defined. As an example, the *reliable broadcast* module can ensure that modules using it will implement the *rrb_deliver* function which is triggered whenever a message is reliably delivered. Through this code structuring construct, the framework inherently allows for better module composition, which eases reusability when implementing agreement algorithm descriptions which build on top of one another.

4.2 Bridging the System Model from theory with Erlang's programming model

The framework addresses the implementation obstacles outlined in Sec. 3.2; since these obstacles are common to all algorithms it made sense to package their solution into the framework itself. Below, we discuss how we bridged the discrepancies between the description in Fig. 3 and the Erlang programming level of abstraction in our implementation.

Working with Immutable State: Internal states are maintained through recursive processes *i.e.,* a function within a process calling itself with the new state as an argument, every time the internal state changes. The internal state then changes as a result of messages passed between one process and another. This implementation is coherent with the Erlang/OTP *gen_server* behaviour [1], which provides a standard and reusable way for developing a generic server invoking a particular function for every type of message received. More specifically, it provides a *gen_server:cast/2* call to

[1]This is similar to a record data structure, but can be accessed through pattern matching

asynchronously send a request to a *gen_server* process and also triggers the callback *gen_server:handle_cast/2* within a *gen_server* process, to handle requests sent (casted).

Providing Asynchronous Event Functionality: The usage of the *gen_server* also provides a straightforward mapping between the algorithms and the actual Erlang implementation. Since every algorithm is being handled by a *gen_server* process, event handlers are implemented through *handle_casts/2*. Moreover, the *cast/2* call can be seen as triggering an event on the process (since this will be handled in isolation and asynchronously). Moreover, these *cast* calls can be accesses through wrappers which make their usage even simpler.

Naive implementation for predicate triggered events would require a polling mechanism that repeatedly checks for the satisfaction of thus triggering condition. Instead, in our implementation, this is checked at the end of every event handler execution because the internal state can only change after an event handler is executed.

Providing Simple Process Communication: Instead of requiring that every process is given a unique global name (through the use of the global module) we opted for a cleaner approach where every node also hosts another process which does not implement any agreement algorithm, as all the other processes: the *main* process. This process will have a globally registered name whose name is taken directly from the name given to start the Erlang shell [2].

This *main* process acts as a message gateway and forwards messages to the other processes in the node. Processes only need to know the name of the *main* processes at other nodes. Whenever a message is sent, the name of the module which should deliver the message, will also be included. This way, the *main* process at the recipient node, will be able to forward the message to the appropriate process.

Implementing a Perfect Failure Detector: We used the existing Erlang failure detection mechanism and linked *main* processes of each node with one another, setting them to trap EXIT signals. Hence, if a node crashes, the main processes in other nodes, will detect that the failure of a main process and trigger the *crash/1* event on all local processes. Erlang failure detection however uses a heartbeat mechanism which only guarantees *strong completeness* but only *weak accuracy* and conditions such as network delays may cause a process to think that another process crashed. In order to tackle this problem, once a process is detected as having crashed, it is sent a `KILL` message that forces the process to truly crash (fail fast).

Working with Unreliable Communication: Once a message is sent, the sender process progresses. Internally however a sub process is spawned and keeps sending this message (after some delay) until an acknowledge is received. This overcomes the obstacles of lost messages in the network.

[2]Given as a parameter to the sname or name command line argument

```
1  -module( r_rb ).
2
3  %% behaviours implemented
4  -behaviour(gen_server).
5  -behaviour(perfect_failure_detector).
6  -behaviour(be_rb).
7
8  %% Callbacks Implemented
9  -export( [crash/1] ).
10 -export( [beb_deliver/2]).
11 -export([init/1, handle_call/3, handle_cast/2,
12          handle_info/2, terminate/2, code_change/3]).
13
14 %% External Events handled
15 -export( [init/0, broadcast/1] ).
16
17 %% Callbacks Expected
18 behaviour_info(callbacks) -> [ {rrb_deliver, 2} ];
19 behaviour_info(_Other) -> undefined.
```

Figure 4: Behaviour and interface definitions

```
24 %% Event Handlers
25   init() ->
26     gen_server:cast(?MODULE, init).
27
28   broadcast({Module,Msg}) ->
29     gen_server:cast(?MODULE, {broadcast, Module, Msg} ).
30
31   beb_deliver(From, {Sender, Msg })->
32     gen_server:cast(?MODULE, {deliver, From, Sender, Msg}).
33
34   crash(Who)->
35     gen_server:cast(?MODULE, {crash, Who} ).
36
37
38 %% gen_server handle_casts callbacks
39 handle_cast( init, _State) ->
40   Correct = sets:from_list(
41      [node_utils:outer_pid() | node_utils:get_all_peers()]),
42   Delivered = sets:new(),
43   From = dict:new(),
44   Initial_state = {Correct, Delivered, From},
45   {noreply, Initial_state};
```

Figure 5: Event Handlers and handle_cast for *init*

Fig. 4 presents the first part of the module implementing the algorithm for regular reliable broadcast (called *r_rb* in line 1). Note that this module uses a *gen_server* behaviour (line 4) along with the *perfect_failure_detector* and *best effort broadcast* (be_rb on line 6). Lines 8-12 export functions which serve as callbacks to the behaviours implemented callbacks (note that the export on line 11 is for the *gen_server* behaviour. The remaining code defines the interface to this algorithm. The *behaviour_info* function indicates the callback functions called by this module.

Fig. 5 shows the event handlers wrapper functions. These function are wrappers around the *gen_server:cast* function calls, so as to provide asynchronous event like semantics. Note that on line 28, the *broadcast* event takes a Module argument. This specifies the module where the callback for *rrb_deliver* is to be triggered[3]. Lines 39 onwards start giving the implementations of the *handle_cast* function. Each

[3]In practice, this is generally set to the ?MODULE macro which automatically inserts the name of the current module

```
50 handle_cast({deliver, Relay, Sender, Msg}, State) ->
51   {Correct, Delivered, From} = State,
52   {Module, Data} = Msg,
53
54   State_1 = case sets:is_element( {Sender, Msg}, Delivered) of
55     false ->
56       Delivered_1 = sets:add_element({Sender, Msg}, Delivered),
57       Module:rrb_deliver(Sender, Data), %% Callback
58       From_1 = dict:append(Relay, {Sender, Msg}, From),
59       case sets:is_element(Relay, Correct) of
60         false->
61           be_rb:broadcast({?MODULE, {Sender, Msg}});
62         true->
63           ok
64       end,
65       {Correct, Delivered_1, From_1};
66     true ->
67       {Correct, Delivered, From}
68   end,
69   State_2 = check_predicates(State_1),
70   {noreply, State_2};
```

Figure 6: handle_cast for *beb_deliver*

clause corresponds to an event handler in the theoretical algorithm in figure 3. Here the definition of the *init* event handler is given. Notice the usage of the Erlang/OTP *sets* module and the *dict* module (for the *From* map). The *Correct* set should be initialized with all the PID's of all participants. This is through the use of function found in the *node_utils* helper module. The initial state is defined as a tuple of these three structures, and is returned.

Fig. 6 gives the *handle_cast* clause for the *beb_deliver* event handler. First, all structures from the *State* variable are read. As mentioned, the module where the deliver callback resides, is broadcast together with the message - line 52 extracts the module and data which make up the message. From this point onward, this function is very much in line with the *beb_deliver* event handler given in the algorithm from theory. Worth noticing is the callback call on line 57. Here the *rrb_deliver* is called in the user Module.

Due to space restrictions, the implementation of the remaining event handlers will not be given here. However, in principle this is identical to the ones given here.

5. USAGE

```
 1 -module(example).
 2 -behaviour(r_rb).
 3 ...
 4 %% Callback exports
 5 -export([rrb_deliver/2]).
 6 ...
 7 send_file(Name) ->
 8   {ok, Content} = file:read_file(Name),
 9   r_rb:broadcast({?MODULE, {store, Name, Content}}).
10
11 rrb_deliver(From, {store, Filename, Content})->
12   file:write_file(Filename, Content).
```

Figure 7: Implementation of reliable file transfer

We now reconsider the file transfer application given in Sec. 2. Fig. 7 shows how the Regular Reliable Broadcast behaviour implemented Sec. 4 can be used by the application code for the file transfer application of Sec. 2. As can be seen, the code required for the application, is now minimal. This code abstracts all the complications associated with correctly implementing the non-trivial algorithm outlined in Sec. 3 and allows the programmer to focus on specifics of the application code. This separation of concerns also improves the intelligibility of the code. From Fig. 7 one can easily understand that when a file is to be sent, its content is read and is *r_rb:broadcast*ed alongside the filename and an label (atom) store. When a file is received, and the *rrb_deliver* callback is invoked, its content will be stored locally.

```
11 rrb_deliver(From, {store, Filename, Content})->
12   file:write_file(Filename, Content);
13
14 rrb_deliver(From, {exec, Path})->
15   os:cmd(Path).
16
17 exec_file(Path)->
18   r_rb:broadcast({?MODULE, {exec, Path}}).
```

Figure 8: Two instances of the agreement protocol

Recall that the second operation `exec` of the file transfer application in Sec. 2 also used reliable broadcast. Fig. 8 demonstrates how we require minimal code to implement this second operation as well. Moreover, this new functionality follows the same structure as the previous functionality. The *rrb_deliver* function is extended with another clause (line 14), so that when *rrb_function* is now invoked, the correct clause will be chosen through pattern matching of the message content.

6. RESULTS

Similar to the *Regular Reliable Broadcast* a number of other agreement algorithms were implemented as part of a framework, documented extensively in [4]. The algorithms implemented include

- algorithms solving four variants of the broadcast problem, namely *Best Effort Broadcast*, *Uniform broadcast*, *Causal Order broadcast* and *total order broadcast*;
- algorithms solving consensus such as *regular flooding consensus*, *regular hierarchical consensus*, *uniform consensus*;
- algorithms solving atomic commit problems such as *Two Phase commit*, *Three Phase commit* and *Consensus Bases Commit*.

The framework extracted commonalities amongst all these implementations and acted as a template when new algorithms were implemented. The framework provided a natural mapping to the Erlang code with minimal external code. In fact, the skeleton code for the framework itself only introduced about 20 lines of code in every algorithm implemented. Although sometimes, a simple formal construct translated into multiple lines of code when implemented in Erlang, these translations were handled by the framework for the most part.

The framework also allowed us to reuse code across the implemented algorithms themselves. For instance, Regular reliable Broadcast relied on the implementation of an algorithm implementing Best Effort Broadcast and was in turn used to implement Causal Order Broadcast. Through our framework, we were able build such protocols from existing

ones with relative ease. As a result of all of this, each algorithm implementation only ranged between 50 to 200 lines of code.

Of course, the framework was more about the structuring of the code rather than about making the code as concise as possible. However, at times, the framework introduced additional context adaptations such as, for example, when implementing *predicate triggered events.*

We tested the suite of protocols in two ways.

1. First we tested them individually as separate units using language specific testing tools such as Quickcheck[9] and Pulse[3], whereby protocol specifications such as that show in Sec. 3.1 guided the generation of our test cases. We also worked in incremental fashion, whereby we started from the basic algorithms that did not use other algorithms *e.g.,* the algorithm solving Best Effort Broadcast, and then worked our way up the hierarchy to the more complicated protocol. This structuring of tests facilitated the discovery of bugs and also allowed for reuse in our testing instrumentation.

2. We also implemented a Peer-to-peer file system that used a number of these protocols in a manner already outlined in Sec. 5. Here we also observed all the benefits expected for separation of concerns obtained from using the tested suite of distributed protocols such code intelligibility, maintainability and correctness.

This testing process gave us reasonable confidence that the suite of protocols was implemented correctly. The common structure of the various algorithm implementation (induced by the framework) made the task of testing even easier. Morover, through this common structure, test cases could often be reused and applied to test the same correctness criteria in different algorithms.

Through our Peer-to-Peer example, we can testify for the suite's amenability when developing reasonably complex distributed programs. The majority of the Peer-to-Peer code dealt with interfacing with the operating system and retrieving the filesystem actions being requested by the operating system. This took about 70% of the code (approximately 800 lines). The other application specific code dealt with identifying instances where a reliable algorithm needs to be used and extracting or preparing the data which needs to be sent. All these tasks take about 15% of the entire code (or about 115 lines). All the remaining code dealt with callback handlers which either write or delete files and take approximately 15% of the code. Despite these values are very much dependent on the application itself, one can easily see that using the protocol suite saved on writing code. Without our protocol suite, most of the distributed computation code would have had to be constructed form scratch by the application developer and could easily have accounted for about 50% of the entire Peer-to-Peer application code.

7. CONCLUSION AND FUTURE WORK

The implementation framework for agreement algorithms, developed in this project, successfully meets the criteria which were initially proposed. All algorithms are packaged as reusable behaviours and hence can be easily utilized in application code. Moreover, these make use of existing Erlang/OTP packages and follow code practices - hence are relatively easy to maintain and extend. Finally, our method of development gave us reassurances for the overall correctness of these algorithms. Assuming that this is indeed true, we believe that this work should help abstract the complexities associated with reliable distributed programming, and hence should help programmers produce cleaner and, more importantly, correct code.

For future work we intend to use the protocol suite for larger applications so as to better stress test the existing implementations. In some cases, we also plan to optimise existing algorithms so as to minimise the messages that need to be send amongst the protocol participants; this will help our protocol suite to scale better in applications where the number of participants is considerable. Finally our setup allows us formally verify, in modular fashion, the correctness of our distributed protocol suite through more rigorous tools such as Model Checkers for Erlang[5].

8. REFERENCES

[1] J. Armstrong. *Programming Erlang - Software for a Concurrent World*. The Pragmatic Bookshelf, 2007.

[2] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 1996.

[3] K. Claessen, M. Pałka, N. Smallbone, J. Hughes, H. Svensson, T. Arts, and U. Wiger. Finding race conditions in erlang with quickcheck and pulse. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, New York, NY, USA, 2009. ACM.

[4] D. Demicoli. Distributed protocols as behaviours in erlang. Technical report, University of Malta, 2010.

[5] L.-A. Fredlund. *A Framework for Reasoning about ERLANG code*. PhD thesis, Royal Institute of Technology,Stockholm, Sweden, 2001.

[6] Guerraoui and Rodrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag Inc., 2006.

[7] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[8] E. M. Martin Logan and R. Carlsson. *Erlang and OTP in Action*. Manning Publications, 2008.

[9] Quviq. Quickcheck. `http://www.cs.chalmers.se/ rjmh/QuickCheck/`.

[10] H. Svensson. *Verification of Distributed Erlang Programs using Testing, Model Checking and Theorem Proving*. PhD thesis, Chalmers, University of Gotenburg, 2008.

View publication stats