

Are High-level Languages suitable for Robust Telecoms Software?*

J.H. Nyström¹, P.W. Trinder¹, and D.J. King²

¹ School of Mathematical and Computer Sciences,
Heriot-Watt University, Edinburgh, EH14 4AS, Scotland
{jann, trinder}@macs.hw.ac.uk

² UK Software & Systems Engineering Research Group,
Motorola Labs, Basingstoke, England
David.King@motorola.com

Abstract. In the telecommunications sector product development must minimise time to market while delivering high levels of dependability, availability, maintainability and scalability. High level languages are concise and hence potentially enable the fast production of maintainable software. This paper investigates the potential of one such language, Erlang, to deliver robust distributed telecoms software. The evaluation is based on a typical non-trivial distributed telecoms application, a Dispatch Call Controller (DCC) measured on a Beowulf cluster. Our investigations show that the Erlang implementation meets the DCC's resource reclamation and soft real-time requirements, before focusing on the following reliability properties.

- **Availability**, e.g. recovery from failures is fast and repeated failures don't reduce post-recovery throughput.
- **Redundancy degree**, e.g. how many simultaneous copies of the system state can be maintained without impairing performance?
- **Resilience**, e.g. achieving a throughput of 101% at 1000% load on 4 processors.
- **Dynamic adaptability**, e.g. the system can be dynamically upgraded by adding nodes without interruption of service.

We critique Erlang's fault tolerance model, arguing that it is low cost, parameterizable and generic. As the Erlang DCC is less than a quarter of the size of a similar C++/CORBA implementation, the product development in Erlang should be fast, and the code maintainable. We conclude that Erlang and associated libraries are suitable for the rapid development of maintainable and highly reliable distributed products.

1 Challenges for Distributed Telecoms Software

The demands of today's telecoms consumer are simply put: they want low-cost, easy to use systems that are reliable and always available. Achieving such high

* The project is in part funded by the UK EPSRC Research Project, GR/R88137 and is a joint venture between Motorola Software and Systems Engineering Research Group at Basingstoke (England) and Heriot-Watt University at Edinburgh(Scotland).

standards of reliability and availability presents a grand challenge to the telecoms industry, especially when the architecture by necessity is distributed, uses an array of different hardware, networks, operating systems, as well as application software. In the face of stiff competition, reducing time-to-market is essential.

Currently many telecoms systems are implemented in C/SDL on real-time operating systems with trends towards using C++/CORBA, JAVA/RMI and UML 2.0 statemachines. Higher level programming language technologies like Erlang [1] potentially offer significant advantages. Development time can be reduced, and maintainability improved because programs are shorter as they specify less low-level detail. Reliability is improved by safe type systems and relatively easy verification, using an interactive proof assistant with an embedding of the language in the proof rules [2]. Clearly the language technology used must also meet the other functional requirements of telecommunication applications, e.g. real-time requirements.

This paper reports the results of an investigation into the effectiveness of Erlang for constructing robust telecoms software. We start by giving a brief introduction to Erlang (Section 2). As a basis for the evaluation we have re-engineered a substantial Dispatch Call Controller (DCC), originally 15K lines of C++/CORBA, as 5k lines of Erlang (Section 3). We show that the Erlang implementation meets the resource reclamation and soft real-time requirements of the DCC.

The primary focus of our investigation is reliability. Availability experiments investigate single, repeated and multiple failures of various system components (Section 4.1). Redundancy experiments investigate how many workers with fully replicated system state can be maintained (Section 4.2). Resilience experiments investigate performance under 100%, 200% and 1000% load (Section 4.3). Dynamic adaptability experiments investigate system throughput as processors are removed and added (Section 4.5). To draw general conclusions about the suitability of Erlang for constructing robust software we critique its fault tolerance model, using illustrations drawn from the DCC (Section 5).

2 Erlang

Erlang is a distributed functional language developed specifically for constructing high reliability telecommunications systems [3]. It is a new software development technology seeking to establish itself in the telecommunications area. To reduce time to market the language is supplied with an Open Telecom Platform (OTP) [4] that includes inter alia: libraries, tools, distributed debugging support, foreign language interfaces and design principles. Erlang is open-source and supported by commercial training courses and technical support, an international user conference together with books and online reference material.

Erlang has been used by a number of large telecommunications companies to construct a wide range of such applications as the first implementation of GPRS for standard packet data in GSM systems [5], and the Intelligent Network Service Creation Environment [6]. The largest application to date is the AXD

301 scalable and robust backbone ATM switch [7], currently utilising up to 32 Processing Elements (PEs). The code comprises over 1.7 Million lines of new Erlang code [1, p.6], 300K lines of mostly-reused C and 8K lines of Java, developed by a team peaking at 50 software engineers [8].

Erlang has several features that facilitate the construction of large distributed real-time systems. The module system allows the structuring of very large programs into conceptually manageable units. Reliable software is constructed using fault tolerance mechanisms, namely timeouts, exceptions and a mechanism where a process can monitor the termination of other processes. Code loading primitives allow the system to be upgraded without stopping the system: an essential requirement for many applications including telephone exchanges. Erlang has a process-based model of concurrency, and processes communicate by message-passing. Erlang supports single-assignment variables, and has an explicit notion of time, enabling it to support soft real-time applications, i.e. where response times are in the order of milliseconds

3 Dispatch Call Controller (DCC) and Platform

To evaluate Erlang we have chosen to re-engineer an existing prototype dispatch call system developed at Motorola Labs, Illinois [9]. Dispatch call processing is a prevalent feature of many wireless communication systems. Managing the call processing with a distributed paradigm enables the processing to be scaled as system usage grows, with work dynamically distributed to the resources available. The essence of the application is a group call manager that generates instances of a group call factory dynamically on the resources available. Each factory generates call handlers to manage individual calls sent to it by the manager. Both C++/CORBA and Java/JINI DCC implementations exist [9].

The DCC requires the following functionalities. Dynamic scalability, i.e. the ability to adapt to use additional resources while the system is running. Resource reclamation, ensuring that once a service instance has terminated, its resources are reclaimed. Fault tolerance and providing continued service despite failures in particular. Soft real time performance: call management mustn't interrupt the call.

3.1 The DCC model

The requirements for the DCC model are derived from the technical report by Lillie [9] and on the functional requirements document from that project [10].

The experiments have been conducted using a model of the DCC service that only deals with regular voice point-to-point calls and hand-offs between the base radio stations. Calls received when the workers are already executing the maximum number of instances, are rejected.

The system may have between 1 and 4 worker nodes running on separate processing elements and 13 processing elements dedicated to the traffic generator and system interface.

The model of the DCC service comprises two distinct parts, the subscribers of the system generating traffic and the fixed-end terminating the service.

The subscribers are simulated by two processes each; the first will issue hand-off messages to the fixed-end notifying it that the subscriber is now associated with a new base radio station, thereby simulating the roaming of the subscriber. The second process setups and generates the voice traffic for the call. The call is initiated by a request from the caller process which waits for the fixed-end setting up the call and routing the traffic; a timer is set when the request is sent modelling that a user would terminate the call if the call is not commenced within a short timespan.

The rates of hand-offs and call requests are higher than one would normally expect from a real system, this is in order to generate large enough volumes of traffic without having an unwieldy number of subscribers in the generator. The requests and the duration of calls are evenly distributed over the expected rates which are: one hand-over every two minutes; one call request every ten minutes; each call last half a minute. During a call the initiating party, which is also the sending party this being simplex traffic, will issue voice data each 90 milliseconds.

The fixed-end is simulated by the framework with a service dealing with both hand-offs and call requests. The service requires one database table to keep track of which subscribers are associated with which base radio station. The hand-offs only result in that the record keeping track of the subscriber is updated; whereas for the calls the service instance will be maintained as a handler for the duration of the call. The handler is maintained for a certain time after the call is completed, should a new call be setup for the same subscriber within a short timespan. The call handler in the fixed-end must also be able to deal with hand-offs during the call.

3.2 Platform

The hardware platform is a 32-node Beowulf cluster, where each node consists of a Pentium-III 530 MHz processor with 256 Mbytes of RAM, interconnected by a switched 100Mbit Ethernet.

The software test platform is made up of a number of subsystems described below and the overall architecture is shown in Figure 1.

Test Management Responsible for starting and controlling the System Management and Traffic Generator subsystems during the test. The Test Manager will also inject the faults into the non-testing subsystems.

Traffic Generator The Traffic Generator sends a sequence of calls to the Service Port and acts as a sink for all messages from service instances to caller.

System Management Responsible for starting, stopping and management of the Service Port and the worker nodes. The System Management subsystem is also responsible for restarting the Service Port for any worker that fails.

Service Port The Port is responsible for starting and maintaining all the interfaces used by the services to communicate with the Workers and relays calls

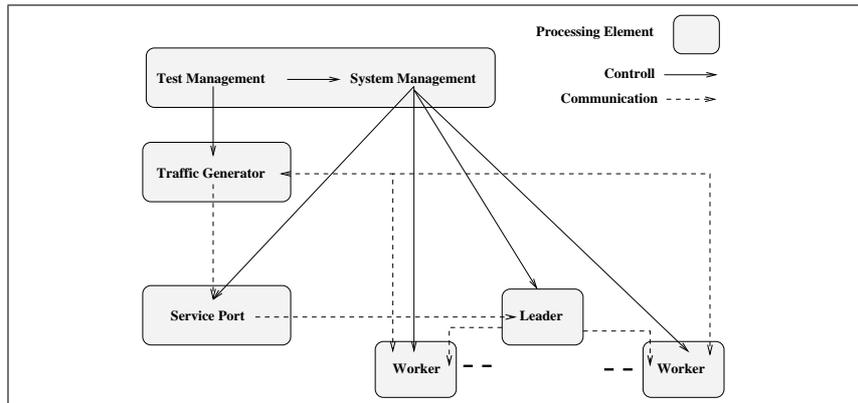


Fig. 1. System architecture

from the subscribers of the services to the Worker responsible for Service Admission acting as gatekeeper.

Worker There is one or more Worker subsystems in the system and they are responsible for executing of the dispatch call handlers. One of the workers is the designated leader of the workers and is responsible for admission control and distribution of calls between the available Workers. The leader is also responsible for redistributing the call handled by a worker should it fail and for restarting the System Management subsystem should it fail. Should there only be the leader it will also act as a normal worker.

The leader of the workers is selected using a distributed leader election protocol based on functionality in a library enabling distributed name registration. Should the Leader fail a new leader will be elected using the same protocol.

4 Experiments

This section reports investigations of robustness properties of the DCC, specifically availability, redundancy, resilience, telecoms characteristics and dynamic adaptability.

4.1 Availability

An important property of any distributed system, that is expected to have a high availability, is that not only should it be able to handle failure of one of the component systems but also repeated and simultaneous failure in several components.

The reason that it is important that the system can deal with multiple components failing is that it is not uncommon that the behaviour of a faulty component may well cause other components to fail before it fails itself; even if the

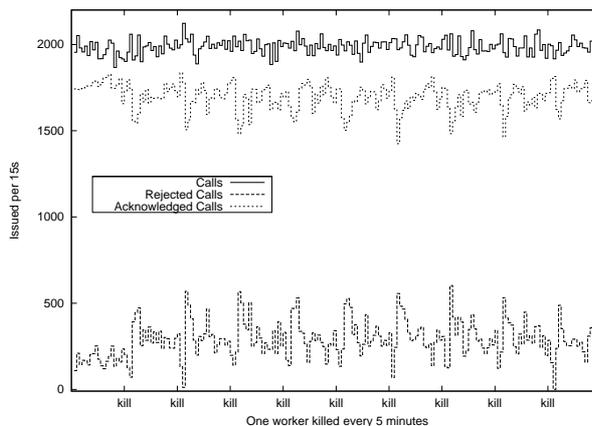


Fig. 2. Repeated failures

other components are functioning entirely correctly except in the case when one of the other components cease to function properly.

To test the system for single failures, repeated failures and multiple failures we have subjected the system to the following tests:

- One worker node in the system is repeatedly crashed (Figure 2);
- Several different nodes are crashed, one after the other, making sure that the node elected leader is among the nodes crashed as well as the node elected to replace the first leader (Figure 3);
- Groups of nodes of increasing size are crashed (Figure 4);
- Repeated and multiple failures of the service instance processes (Figure 5).

The reason we have chosen to crash nodes as a means of inducing failures is that the various processes comprising the worker node framework are setup to terminate should any one of the main processes fail; the remaining cases are simulated by individual service instance processes failing.

As the graphs in Figure 2 to Figure 5 shows the system recuperates and regains its former capacity within a reasonable time even in the case where all but one of the nodes has failed.

4.2 Redundancy

Reliable scalable systems are often constructed by composing units with internal replication. For example in the AXD 301 uses pairwise replication with up to 16 pairs of processors [7]. Where some systems replicate hard and software wholesale, in Erlang only the parts of the state required to recover from a failure is maintained in the Mnesia distributed database [11]. While increasing the degree of replication makes the system more reliable (i.e. more nodes can fail without loss of data), the cost of maintaining the replication is quadratic in the degree of replication.

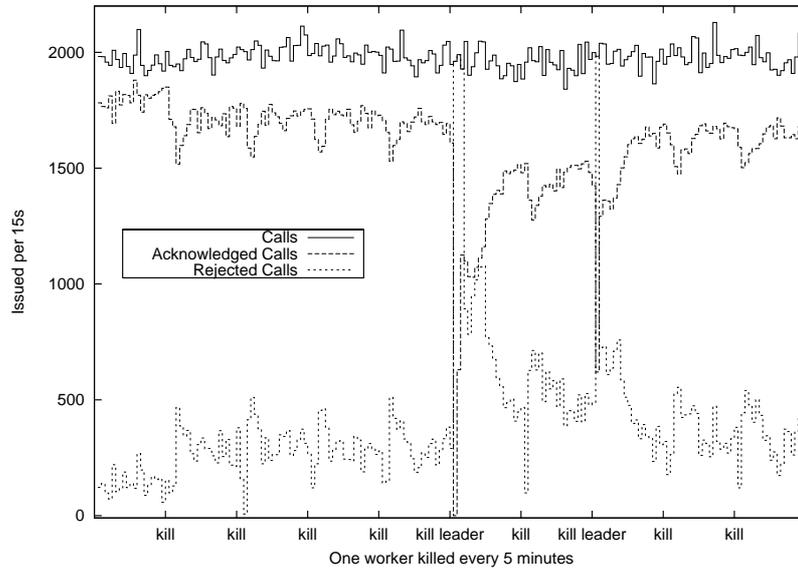


Fig. 3. Several failures

We investigate highest degree of redundancy that can be provided in the DCC without impairing performance. This is investigated by comparing the number of service calls handled per time unit at 100% load with a varying number of fully-replicated worker nodes. For DCC we define 100% load as the number of subscribers that are handled with fewer than 1% of the requests rejected and fewer than 0.1% of message traffic undelivered. Recall that subscribers stochastically generate calls and hand-offs, and hence the relative measure of load.

The service load is provided by traffic generators simulating service calls from subscribers of the services, where the generator generates the number of subscribers per processing element times the number of processing elements.

The result of the experiment is shown in Figure 5, where Figure 5 shows the increase in calls per second on 1 to 8 nodes. The curves plotted are as follows.

X%Load The number of calls handled by the service at X% load on n nodes with n-times replication.

Ideal The results we would get if we had linear speedup with increasing workers.

These results show a reasonable speedup of 3.7 times with 4 worker nodes, and that the cost of replication is quite acceptable up to four time replication. Other experiments show that without any replication the system scales well, giving a speedup of 13.95 on 16 worker nodes.

This shows that we can, without impairing the performance, have fourfold redundancy and that we can build systems of larger number of nodes with them grouped in redundancy foursomes.

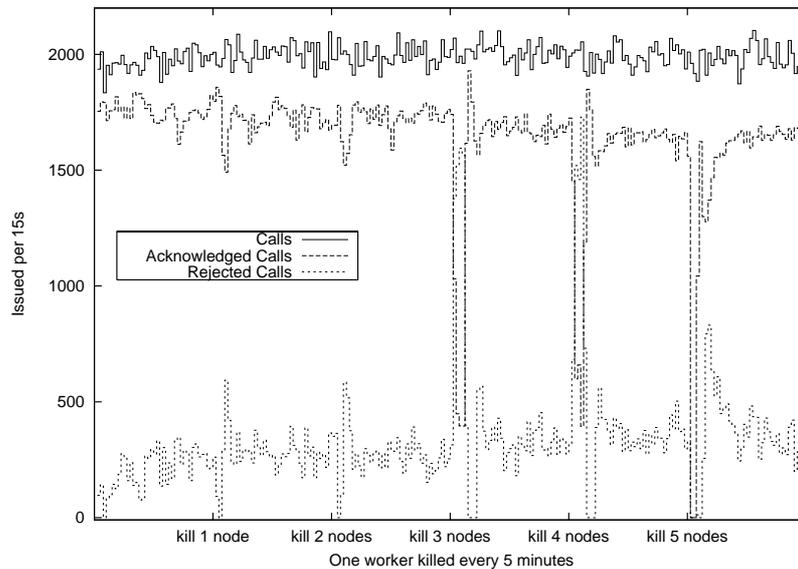


Fig. 4. Multiple failures

4.3 Resilience

Overloading the system should result in a graceful degradation of performance. In the context of the DCC an overload is more simultaneous service calls than the system can handle. Like most telecoms systems the DCC controls load by declining requests at the interface.

Resilience is investigated by subjecting configurations of the system to 100%, 200% and 1000% load with a varying number of worker nodes. This experiment uses the same notion of load and traffic generator as in the scalability experiment.

Figure 5 shows several interesting features. Throughput at 1000% is always less than throughput at 200%. Surprisingly, up to 3 nodes there is a small increase in both request and message throughput for both 200% and 1000% load. Thereafter both 200% and 100% result in lower throughput. The reason that an overload at a small number of nodes gives a small increase in throughput is that at 100% load there are times when one or more of the worker nodes execute less than the maximum of service instances.

We have also exposed a one worker system to 50'000% load and the throughput was still 110%. Limitations of the load generation technology prevent measurements of larger systems at such exceptionally high loads.

4.4 Telecoms characteristics

Two important aspects of telecoms systems are timeliness and resource reclamation, where the time constraint exhibited by the DCC fall into the category

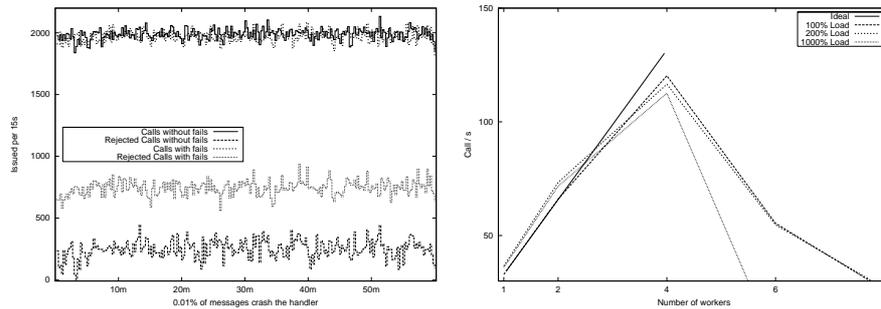


Fig. 5. Service instance failures and Increase requests handled

of soft real time, meaning that although a very small number of calls may fail entirely, however in all unfailing calls the call control and service payload must be delivered on time. In order to ensure that we conform to the time constraints we have set timers in the traffic generators, and we monitor that voice messages are delivered. During all the tests these properties are checked.

In high availability systems the reclamation of resources is vitally important since any leakage of a resource means that sooner or later the system will crash from lack of that particular resource. One important part of resource reclamation is the management of memory allocated for the call, but since Erlang has automatic memory management this is trivial. The other main problem would be the allocation of central system resources like database entries or tables; all such resources are accessed through a common mechanism where the resources allocated to a service instance are noted in the distributed table entry for the service instance and all resources not deallocated when the instance terminate is automatically deallocated. We have tested this by running the system over long periods of time without any increasing usage of resources.

4.5 Dynamic adaptability

To provide high availability telecoms systems should adapt quickly to changing demands, such as the resources required. It is not only important that computational and other resources can be added quickly, and without major performance costs during the adaption process, but also that the same resources can be removed when they are no longer needed.

System behaviour as computational resources are added and removed is evaluated as follows. The system is started with 4 worker nodes and nodes are removed until only the leader node remains and then the nodes are added back again until all nodes have been added. The system has been exposed to a 100% load for 4 worker nodes, using the same notion of load as in the scalability experiment.

The Figure 6 shows the result of the experiment and illustrates the following properties.

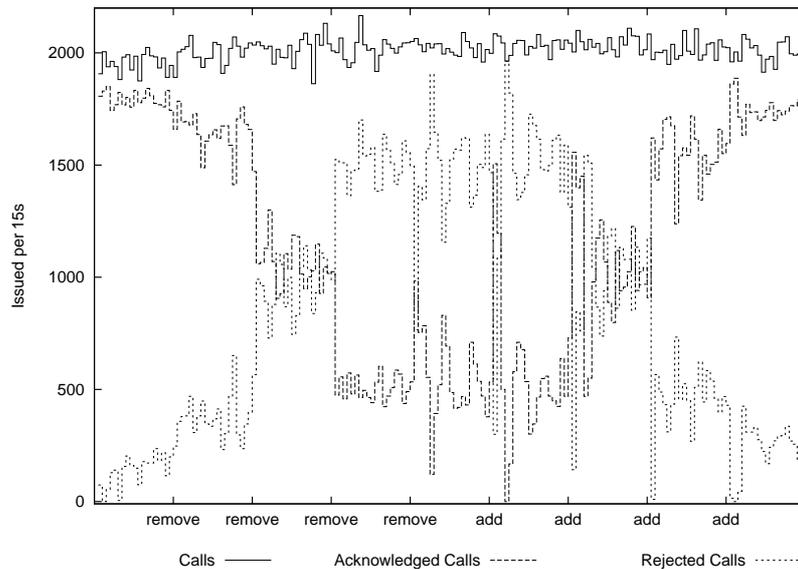


Fig. 6. Dynamic adaptability

- The first half of the graph shows near linear decrease in performance when a node is removed. For example, the approximate number of acknowledged calls per 15s falls from 1800 on 4 nodes to 950 on 2 nodes and 500 on 1 node.
- The second half of the graph shows near linear increase in performance when a node is added. For example the approximate number of acknowledged calls per 15s rises from 500 on 1 node to 950 on 2 nodes and 1800 on 4 nodes.
- The cost of adding/removing a node is small: any fall in throughput as nodes are added and deleted is small.

5 Evaluating the Erlang Fault Tolerance Model

This section critiques the aspects of Erlang fault tolerance model used in the DCC. In implementing the prototype application we have relied on four aspects of Erlang to achieve a fault tolerant system:

- Process encapsulation;
- the ability to monitor other processes and Erlang-nodes for failures;
- OTP library implementations of common supervising design patterns;
- and the OTP distributed database Mnesia implemented in Erlang.

Erlang is a language with true light-weight processes that enables the programmer to use a separate process for each of the tasks the programme has to perform, and yet they are most similar to operating system processes with a non-shared

state and independent runtime system managed scheduling. The processes communicate by asynchronous message passing, where the order of messages is only guaranteed for messages from the same sender. This ensures that the process act as a proper encapsulation of computation and a process can only be influenced by another process if it explicitly allows it by receiving messages from that process.

Encapsulation is vital to achieve fault tolerant software, since for the system to be able to handle a faulty component (process) may not unrestrictedly influence other components (processes), otherwise the faulty component may compromise the very components that is supposed to deal with the fault. To encapsulate the faulty behaviour it is also important to minimise the effects of a fault, since if a process controls how it may be influenced it makes it that much easier to detect malign influences.

In order for processes to be able to deal properly with failures in other processes they have to be able to not only determine when such a failure has occurred, but also why, the monitoring facilities of Erlang allows for this information. The monitoring in Erlang is achieved via a bi-directional link that is explicitly set up between processes. When a process fails through an uncaught exception the fact that it has failed and the exception is propagated to all the linked processes. A linked process can choose if it will perform the default action upon receiving the information that a linked process has failed, which is to fail itself, or the process can receive the information as a special message in the message queue.

The notion of links allows the Erlang programmer to properly separate the two concerns of normal execution in his programme and the handling of errors, where one process deals with each aspect. When a programme consists of several tasks each dealt with by one or more processes one comes frequently to the situation where groups of processes with dependencies have to be monitored. A convenient way of organising such groups of process is into supervision hierarchies with several layers of supervising processes being able to deal not only with the failures of individual processes but also with their interdependencies. This common design pattern has support in a library called the supervisor [1].

When dealing with a failure in a component it is often necessary to have a record of the vital part of the failed components state. The state may in the actual implementation be spread over several cooperating processes; this is called a persistent storage since it has to persist the failure of the component. There are many ways of supplying persist storage, e.g., saving logs of state changes to files. In Erlang one common way is to store such information in the distributed database Mnesia that comes with the system. Mnesia is a table-based distributed database implemented entirely in Erlang [11].

5.1 Concise and Clear

The fact that we have true operating system style processes affects the size and clarity of the code in two ways. Firstly, when dealing with systems that have a large natural concurrency like telecoms it is much simpler to implement

the concurrent parts as separate processes. The support for processes in Erlang means that very little explicit management of these processes is required. The code size and more importantly the clarity of the code is much influenced by the fact that Erlang has automatic memory management; memory management is normally a great source of complication and errors.

Secondly, the separation between normal execution and fault handling means that the normal execution will not be cluttered up with code dealing with abnormal and erroneous situations and likewise for the fault handling part of the code. This avoids the burden of defensive code where one has to perform torturous and involved testing of every return from a function that may fail and deal with every possible different failure at that point. The failure testing and handling code spread all over the programme when programming defensively is often very similar, and often results in the dangerous habit of cut-and-paste programming. When the common parts of fault handling are separated into a fault handling process one can do without the replication and the result is much smaller and indeed simpler.

ERLANG				C++			
Part	LOC	Percentage	Modules	Part	LOC	Percentage	Classes
Total	4882	100%	38	Total	14857	100%	
Platform	2994	61%	26	C++	14774	99.4%	36
Testing	1741	36%	11	IDL	83	0.4%	15
Service	147	3%	1				

Table 1. ERLANG and C++ DCC Code Sizes

The size of the ERLANG and C++/CORBA DCC are reported in Table 5.1. The number is divided into Platform: which are the parts that are generic and not specific to the DCC service; Testing: which are the parts that are solely involved in the testing and statistics gathering; Service: which are the parts specific to the DCC. The earlier C++ prototype consisted of 15KLOC of C++ and 83LOC of IDL.

5.2 Low Cost Reliability

We argue that the fault tolerance of Erlang is inexpensive in two ways. Firstly the low cost of processes in Erlang means that the cost in resources of having additional processes monitoring fault handling is very low. This can be compared to the cost of performing pervasive testing of all return values or encapsulation performed with proper operating system processes.

The second way in which the Erlang fault tolerance model is inexpensive is in the design and realisation of the system, since in order to achieve a fault tolerant system one has to take into consideration the possible fault and how they can be trapped and dealt with in the design. Being able to separate the issue of fault tolerance gives a simpler design and simplifies the realisation of the design, we

do not have to worry about all the point in the system an error may occur, but what conceptually different errors may occur and how we may best deal with them [1].

5.3 Parameterizable and Generic

One very expensive part of fault tolerance in distributed systems is the need to maintain a consistent persistent storage to deal with failures in components on other physical units. It is vital to keep the information needed to recover from a distributed failure to a minimum, although it is always going to be an exponential growth in communication with respect to the number of copies you have of the information. The usage of Mnesia makes it easier to customise the trade off between the number of copies of the distributed data using fragmented database tables. Mnesia allows explicit control of how many copies, in memory or on disk, each table fragment should have. It is thus possible to make it a configuration parameter how many copies should exist, and this parameter may be dynamically changed during the execution of the system.

The use of a high-level language gives the usual benefit that much of the code is actually application neutral and can be reused in other applications, this is even true of the fault tolerance mechanism we use. The most obvious proof, that much of the fault tolerance code is application neutral, is the use of the supervisor library module that is itself implemented in Erlang.

6 Discussion

Summary This paper reports the investigation into the effectiveness of Erlang for constructing robust telecoms software. We have constructed and measured a typical telecoms application, the DCC. The Erlang implementation meets the functional requirements of the DCC: 99.9% of all voice messages are delivered on time; resources are automatically reclaimed as memory and processes are automatically managed.

Availability experiments show that recovery is fast and the implementation can handle single, repeated and multiple failures of various system components without significant reduction in post-recovery throughput. Redundancy experiments show that the system can provide fourfold redundancy without significant performance penalties.

Resilience experiments show that performance does not significantly degrade under 200% over 1000% load. This would ensure that the system would survive massive overloads, for example from a denial of service attack. Dynamic adaptability experiments show that performance degrades linearly as processors are removed from the system and likewise upgrades linearly as processors are added.

Conclusions To draw general conclusions about the suitability of Erlang for constructing robust software we have critiqued its fault tolerance model. We

argue that it is well-suited to the development of robust telecoms software as it is relatively low cost, parameterisable and generic.

Our work gives some evidence of the conciseness of Erlang: the Erlang DCC is less than a quarter of the size of a similar, but not identical, C++/CORBA implementation. Conciseness reduces both development and maintenance time and costs. We conclude that Erlang and associated OTP libraries are suitable for the rapid development of maintainable and highly reliable distributed products.

Future Work Following on these initial experiments the teams at Motorola Basingstoke Labs and Heriot-Watt will re-engineer part of a product. We are currently discussing the choice of application with a development team within Motorola.

We also aim to assess the relative merits of other languages for distributed telecoms products by comparing the current DCC implementation in Erlang with the existing in C++/CORBA and JAVA/CORBA and an implementation in Glasgow Distributed Haskell [12] that we have developed.

References

1. J.Armstrong: Making reliable distributed systems in the presence of software errors. PhD thesis, Department of Microelectronics and Information Technology, Royal Institute of Technology, Stockholm , Sweden (2003)
2. Fredlund, L., Gurov, D., Noll, T., Dam, M., Arts, T., Chugunov, G.: A verification tool for erlang. *International Journal on Software Tools for Technology Transfer* **4** (2003) 405–420
3. Armstrong, J., Virding, R., Wikström, C., Williams, M.: *Concurrent Programming in ERLANG*. 2nd edn. Prentice Hall (1996)
4. Torstendahl, S.: *Open Telecom Platform*. Ericsson Review (1997)
5. Granbohm, H., Wiklund, J.: *GPRS - General Packet Radio Service*. Ericsson Review (1999)
6. Hinde, S.: Use of ERLANG/OTP as a Service Creation Tool for IN Services. In: *Proceedings of the 6th International ERLANG/OTP Users Conference (EUC'00)*, Ericsson Utvecklings AB (2000)
7. Blau, S., Rooth, J., Axell, J., Hellstrand, F., Buhrgard, M., Westin, T., Wicklund, G.: AXD 301: A new generation ATM switching system. *Computer Networks* **31** (1999) 559–582
8. Wiger, U.: *Industrial-Strength Functional programming: Experiences with the Ericsson AXD301 Project*. In: *IFL'00, Aachen (2000) Presentation Only*.
9. Lillie, R.: *Implementing dynamic scalability in a distributed processing environment*. Technical report, Motorola Labs, Schaumburg, Illinois (1999)
10. Rittle, L.: *Distributed Dispatch Architecture Project (Functional Requirements)* (1998)
11. Mattsson, H., Nilsson, H., Wikstrom, C.: *Mnesia - A distributed robust DBMS for telecommunications applications*. In: *First International Workshop on Practical Aspects of Declarative Languages (PADL'99)*. (1999) 152–163
12. Pointon, R., Trinder, P., Loidl, H.W.: *The Design and Implementation of Glasgow distributed Haskell*. In: *IFL'00. LNCS 2011, Aachen, Germany (2000)* 101–116