High-level Distribution for the Rapid Production of Robust Telecoms Software: Comparing C++ and ERLANG*

J.H. Nyström P.W. Trinder D.J. King

Abstract

Currently most distributed telecoms software is engineered using low and mid-level distributed technologies, but there is a drive to use high-level distribution. This paper reports the first systematic comparison of a high-level distributed programming language in the context of substantial commercial products. Our research strategy is to reengineer some C++/CORBA telecoms applications in ERLANG, a high-level distributed language, and make comparative measurements.

Investigating the potential advantages of the high-level ERLANG technology shows that two significant benefits are realised. Firstly, robust configurable systems are easily developed using the high-level constructs for fault tolerance, and distribution. The ERLANG code exhibits *resilience*: sustaining throughput at extreme loads and automatically recovering when load drops; *availability*: remaining available despite repeated and multiple failures; *dynamic reconfigurability*: with throughput scaling near-linearly when resources are added or removed. Secondly, ERLANG delivers significant productivity and maintainability benefits: the ERLANG components are less than one third of the size of their C++ counterparts. The productivity gains are attributed to specific language features, for example, high-level communication saves 22%, and automatic memory management saves 11%—compared with the C++ implementation.

Investigating the feasibility of the high-level ERLANG technology demonstrates that it fulfils several essential requirements. The requisite distributed functionality is readily specified, even although control of low-level distributed coordination aspects is abrogated to the ERLANG implementation. At the expense of additional memory residency, excellent time performance is achieved, e.g. three times faster than the C++ implementation, due to ERLANG's lightweight processes. ERLANG interoperates at low cost with conventional technologies, allowing incremental reengineering of large distributed systems. The technology is available on the required hardware/operating system platforms, and is well supported.

Keywords: Distributed Systems, ERLANG, Telecom Software

*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, Basingstoke and Heriot-Watt University, Edinburgh.

1 Introduction

Telecommunication systems are amongst the most challenging to design and realise since they integrate distributed resources under soft real-time constraints, and must be highly available and reconfigurable. Since the telecommunications sector is highly competitive with rapidly evolving technology, new products require the least possible time to market, and need to be readily extended and maintained.

For the distributed software component that is the focus of this paper, there are three levels of technologies: low-level distribution like sockets or MPI, mid-level distribution like Java/RMI or CORBA, and high-level distribution like UML 2.0 State Machines [23], SDL [22], or high-level distributed languages [8, 1, 25]. Low-level distribution technologies still dominate current telecommunications products, but mid-level technologies are gaining ground. High-level technologies in the form of modelling languages such as SDL, have long been used for design. With modelling language tools acquiring ever more sophistication, however, there is a strong movement towards high-level techniques, and especially model driven development.

This paper evaluates high-level distributed languages in comparison with mid-level technologies like CORBA, in conjunction with C++. High-level languages like ERLANG [1], or Glasgow distributed Haskell (GdH) [25] automatically manage many distributed coordination aspects making the programs concise, with the potential for rapid development and improved maintainability. Moreover, the languages have high-level distributed coordination and sophisticated fault tolerance, facilitating the construction of robust, reconfigurable systems. Significantly, many of the coordination aspects automatically and correctly managed by the language implementations are some of the most critical and difficult issues for distributed software, like storage management, data marshalling communication and fault tolerance.

1.1 Research Goals

This paper investigates whether ERLANG high-level distributed language technology can deliver the potential benefits for realistic telecoms software development, by considering the following research questions.

Q1 Can robust, configurable systems be more readily developed? Does the fault tolerance provided by the language readily enable the development of systems with resilience, high availability and reconfigurability?

Q2 Can productivity and maintainability be improved? We argue that productivity and maintainability are crucially related to code size: shorter programs are faster to develop and easier to maintain. So are the high-level programs more concise, and what is the potential for reuse?

We further investigate the feasibility of the Erlang high-level distributed language technology for realistic telecoms software development by considering the following research questions.

Q3 Can the required distributed functionality be specified? As control of low-level distributed coordination aspects has been abrogated to the language implementation, can the programmer still specify the required functionality?

Q4 Can acceptable performance be achieved? Typically automatic coordination management incurs space and time penalties. Despite these costs, can a high-level language implementation meet the time and space performance requirements of telecoms applications?

Q5 What are the costs of interoperating with conventional technology? Many distributed systems are large, and monolithic reengineering is prohibitively expensive. A new technology that can interoperate with existing technologies allows incremental reengineering, providing that the performance penalties of the interaction between components using different technologies are small.

Q6 Is the technology practical? Pragmatics are a key issue in software technology selection. For example is the technology available on the product hardware/operating system platform(s)? Are suitable libraries available? How well is it supported?

1.2 Research Strategy

Our research strategy is to reengineer some C++/CORBA telecoms applications in Erlang, and make comparative measurements of both implementations for *Time and Space Performance* (Section 5), *Robustness* (Section 6) and *Software Productivity* (Section 7).

Realistic comparisons of software technologies are hard for a number of reasons. It is well known that software development costs do not scale linearly with size, and it is simply too expensive to duplicate large systems, e.g. those with more than 100K lines of code. Worse still, comparing just one system is not sufficient, as while that system may be a good match for a particular technology, many other systems in the same application area are not. Moreover, a comparison must either be based on systems constructed by different teams of software engineers, or the same team will benefit from building the first system when building the second.

The diversity issue is addressed a limited way by reengineering and measuring two C++ telecoms applications, and by considering the results reported for other components in related work (Section 8). We address the issue of scale by reengineering a medium-scale (15K line) Dispatch Call Controller (DCC) [16] (Section 4.2), and a smaller (3K line) Data Mobility (DM) component that is closely integrated with five other components of a base radio network (Section 4.1). Moreover, our results confirm those of a more superficial study of an

extremely large system: 1M lines of code (Section 8). To minimise the learning effect both Erlang implementations are engineered by a programmer not involved in the original Motorola C++/CORBA implementations. For comparability, we assume that the Erlang and C++/CORBA implementers are similarly expert with the technologies used.

1.3 Novelty

We believe the investigation reported here is the first systematic comparative evaluation of a high-level distributed programming language in the context of substantial commercial products. In earlier work we have proposed the current investigation [19], the results of which are reported here. An investigation into the robustness of the DCC is reported in [20], effectively a thorough exploration of research question **Q1**. This paper reports a far broader range of research questions **Q1 - Q6**, and in addition covers the robustness of both the DCC and the DM. Other related work is covered in Section 8.

2 Challenges for Distributed Telecoms Software

The telecoms sector is rapidly growing, with new devices and technologies appearing almost daily. This adds to the complexity of telecoms systems, which by their very nature have a distributed architecture, an array of different hardware, operating systems, networks, in addition to application software. On-time delivery and quality are priorities to keep and win more customers against stiff competition. For example, failing to get the latest mobile phone into the shops before Christmas, could lose market share. Poor quality can have a long-term detrimental effect on brand image, in addition to short-term financial losses, e.g. when a faulty device is recalled. Reliability and availability are key aspects of software quality. Customers don't want their phone to crash, and they want the network infrastructure to be always available. For example, when there was a surge in demand after the London bombings on 7 July, 2005 many mobile networks couldn't cope, and had to shut down or drop calls. This was at great expense to network providers. Typically, a telecoms provider will aspire to the 5-nines of 99.999% availability, which equates to a downtime including maintenance of no more than 5 minutes and 15 seconds in one year. This is rarely achieved, however.

2.1 Technical Challenges

The intention of constructing high-availability software is being addressed by pursuing the following specific technical challenges. *High Level Programming*: using high level programming paradigms in application development releases the programmer from dealing with awkward, low level, technical issues such as memory management and communication details. *Correctness*: telecoms systems are typically too large for the correctness to be shown using formal

proof. Hence, the importance of thorough testing that can take over 50% of the software lifecycle. Additionally, abstraction can help with correctness, since it is easier to demonstrate properties or model check, if the specification or implementation is given in a high-level formal notation. *Fault tolerance*: most downtime is caused not by hardware faults, but by system and application software failure. Recovering from a software crash, or processor failure, improves availability. *Maintainability*: which includes both debugging existing systems, and adding new features.

Application software for telecoms systems have particular requirements that also pose challenges, including the following. *Managing multiple interactions*: application software needs to be expressive enough to manage the interactions between multiple geographically-distributed components, i.e. hardware, software, networks, and operating systems. *Soft real-time*: systems need to respond and react without delay to new requests, often a time-bound for a computation is necessary. *Scalability*: systems need to adapt to cope with increased demand, by the incremental addition of hardware. *Resilience:* the system performance should downgrade gracefully when overloaded. *Dynamic reconfigurability:* to ensure high availability the system has to be able to adapt dynamically to both software and hardware upgrades.

2.2 Current Technologies

Currently many distributed telecoms systems are implemented in C with SDL on real-time operating systems with trends towards using C++/CORBA, JAVA/RMI and UML 2.0 State Machines. Higher level programming language technologies like Erlang [1] are attractive because of the potential to reduce development time, and improve reliability and maintainability. Clearly the language technology used must also meet the other functional requirements of telecommunication applications, e.g. real-time requirements.

3 Erlang and other Distributed Functional Languages

3.1 Distributed Coordination Levels

A distributed system specification must include both an algorithmic aspect, i.e. a correct and efficient algorithm describing *what* to compute, and a coordination aspect specifying *how* to organise the computations across the processors. Distributed coordination typically includes partitioning the program into tasks, placing the tasks on the processors, communicating between and synchronising tasks, and both detecting and recovering from failures. The computational aspect of a distributed program may be specified at a range of levels of abstraction, e.g. relatively low level like assembler or C, or at a high-level like Prolog or Haskell98.

Like the computation aspect, the coordination aspect of a distributed system may be specified at a range of levels of abstraction. At the lowest level the programmer manages all coordination explicitly, e.g. using `send/receive` and `fork/join`. Example technologies providing low-level distributed coordination include sockets and the PVM or MPI communication standards [7, 18]. Mid-level coordination abstracts over several low-level coordination actions, e.g. a remote procedure call (RPC) combines a `send`, `fork`, `receive` triple. Example technologies providing mid-level distributed coordination include Java/RMI and CORBA. High-level coordination aims to be yet more abstract, e.g. modelling distributed processes as state machines. Example technologies providing high-level distributed coordination include UML 2.0 State Machines [23], SDL [22], and high-level distributed languages like Facile [8] and Erlang.

High-level distributed languages like Erlang provide high-level communication, sophisticated fault tolerance mechanisms and automatic storage management. Hence, it is potentially easier to engineer robust software, programs are shorter and hence faster to develop and easier to maintain. The high-level language requires a sophisticated implementation to automatically control many low-level coordination aspects, e.g. synchronisation and data marshalling. As we shall see, relatively few of these sophisticated language implementations have been constructed.

Distributed coordination may be integral to a distributed programming language, e.g. Java/RMI, or provided by an external library, e.g. CORBA or MPI.

3.2 Distributed Functional Languages

In addition to Erlang, a number of distributed functional languages have been constructed with a range of models of processes and communication e.g. Kali Scheme [3], Facile [8], OZ [10], and Glasgow distributed Haskell (GdH) [25]. Almost all the languages are research languages used to investigate high-level distributed coordination integrated with the language. Consequently, these languages are available on few hardware/operating system platforms and have limited support in the form of libraries and tools. In contrast Erlang is a production language designed to aid the rapid production of robust distributed systems. It is available on a range of hardware/operating system platforms and supported by the OTP tools and libraries.

3.3 Erlang

Erlang is a distributed functional language originally developed in Ericsson for constructing highly reliable telecommunications systems [1]. The language has integrated distribution, including first-class processes, sophisticated fault tolerance mechanisms, automatic storage management i.e. garbage collection, soft real-time support, and sophisticated availability support e.g. hot-loading hardware and software upgrades into a running system.

Erlang has been used by a number of companies to construct a wide range of applications, primarily in the telecoms sector, but increasingly in other sec-

tors, e.g. banking. Examples include the first implementation of GPRS for standard packet data in GSM systems [9], and the Intelligent Network Service Creation Environment [11]. The largest application to date is the AXD 301 scalable and robust backbone ATM switch [2], currently utilising up to 288 Processing Elements (PEs). The code comprises over 1.7 Million lines of new ERLANG code [13, p.6], 300K lines of mostly-reused C and 8K lines of Java, developed by a team peaking at 50 software engineers [29].

ERLANG has several general 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. 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. The following subsections outline the specific features of ERLANG that impact the comparisons in Sections 6 and 7.

3.3.1 Fault Tolerance

The ERLANG reliability philosophy is to separate the functionality and error-handling concerns. That is, the programmer writes simple code for the successful case that may fail, raising an exception. The key to this "let it fail" ethos is that the language incorporates first class processes that can fail without damaging other processes. True processes, while common in operating systems, are extremely unusual in production programming languages. A common reason for a failure is a timeout and the exception raised may be handled within a process by an exception handler, or by a monitoring process.

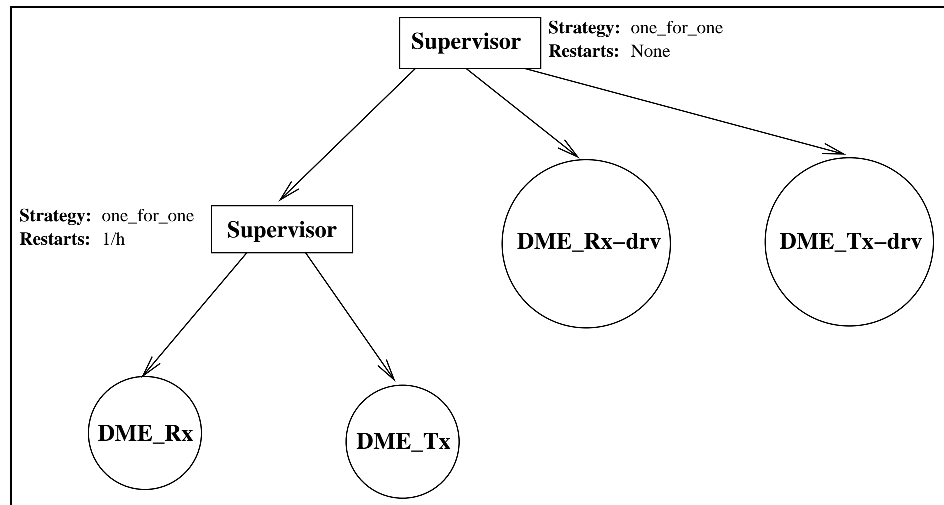

Figure 1: ERLANG/C DM Supervision Tree

The monitoring of one process by another is sufficiently common that it is encapsulated by the *supervisor behaviour* [13]. An ERLANG behaviour is a

high-level distributed coordination abstraction. In the supervisor behaviour the supervising, or parent, process spawns child processes and declares a number of coordination aspects. An important aspect is the action to perform in the event of a failure, e.g. restart the child process, kill the child process, kill all the child processes. A second important aspect is the frequency of failures to be tolerated, e.g. one per hour. As the supervised processes may supervise other processes, a supervision tree can be constructed.

Figure 1 shows the supervision tree for the ERLANG/C DM component described in Section 4.1. In this tree, Supervisor 2 will restart either of the ERLANG receiver or transmitter processes at most once an hour. Supervisor 1 will fail gracefully if supervisor 2 fails or either of the C drivers fail, reflecting the fact that it has no way of restarting the C drivers. This supervision tree provides much of the DM robustness, and is less than 10 source lines of code.

```
sz_dme_dmtx:cast(device_info)
```

Figure 2: ERLANG DM Communication

3.3.2 High-Level Communication

Communication between ERLANG processes is high-level asynchronous message passing. The communication mechanisms provide automatic data marshalling, error detection, communication and synchronisation. Figure 2 and Figure 3 give a dramatic comparison of the same communication in ERLANG and in C++. An ERLANG `cast` is a point-to-point send primitive. The C++ version contains considerable amounts of data marshalling and defensive code, e.g. lines 30-35 detect and report an error. The ERLANG version crucially relies on automatic error detection, and that the failure will be handled elsewhere, most probably by a supervising process.

3.3.3 Automatic Memory Management

It is easy to make errors when explicitly managing memory, moreover space leaks are hard to detect and correct, and cannot be allowed in long running telecoms applications. Like many modern programming languages, ERLANG provides automatic memory management, supported by garbage collection. This both relieves the programmer from specifying a significant and awkward aspect of the program, and improves reliability be guaranteeing safe storage management. However, even with garbage collection, some space leaks may need to be explicitly detected and eliminated.

ERLANG programs typically contain no explicit storage management, and hence there is none in Figure 2. In contrast, lines 9 and 13 of the C++ code in Figure 3 calculate a size and allocate an object of that size.

```
1  void DataMobilityRxProcessor::processUnsupVer(void)
2  {
3     MSG_PTR             msg_buf_ptr;
4     MM_DEVICE_INFO_MSG  *msg_ptr;
5     RETURN_STATUS       ret_status;
6     UINT16              msg_size;
7
8     // Determine size of ici message
9     msg_size = sizeof( MM_DEVICE_INFO_MSG);
10
11    // Create ICI message object to send to DMTX so it sends a Device Info
12    //   message to Q1 and Q2 clients
13    IciMsg ici_msg_object( MM_DEVICE_INFO_OPC, ICI_DMTX_TASK_ID, msg_size);
14
15    // Retrieve ICI message buffer pointer
16    msg_buf_ptr = ici_msg_object.getIciMsgBufPtr();
17
18    // Typecast pointer from (void *) to (MM_DEVICE_INFO_MSG *)
19    msg_ptr = (MM_DEVICE_INFO_MSG *)msg_buf_ptr;
20
21    // Populate message buffer
22    SET_MM_DEVICE_INFO_DEVICE_TYPE( msg_ptr, SERVER);
23    SET_MM_DEVICE_INFO_NUM_VER_SUPPORTED( msg_ptr, NUM_VER_SUPPORTED);
24    SET_MM_DEVICE_INFO_FIRST_SUP_PROTO_VERS( msg_ptr, PROTO_VERSION_ONE);
25
26    // Send message to the DMTX task
27    ret_status = m_ici_io_ptr->send(&ici_msg_object);
28
29    // Check that message was sent successfully
30    if (ret_status != SUCCESS)
31    {
32        // Report problem when sending ICI message
33        sz_err_msg( MAJOR, SZ_ERR_MSG_ERR_OPCODE, __FILE__, __LINE__,
34                    "DataMobilityRxProcessor processUnsupVer: failure sending "
35                    " device info message to DMTX");
```

Figure 3: C++ DM Communication

3.3.4 Tools and Support

To help reduce time to market ERLANG is supplied with the Open Telecom Platform (OTP) libraries [28]. The OTP includes, *inter alia*, libraries, design principles, and productivity, profiling and debugging tools. Example libraries include support for HTTP, FTP, SSL, SSH and TCP/IP protocols, for SNMP agents, CORBA and the H248 protocol stack. There is an Abstract Syntax Notation One (ASN.1) compiler [5]. Database support includes an ODBC implementation and Mnesia, a bespoke in-memory distributed database.

A compiler [14] and a bytecode interpreter are both available open source for ERLANG. The language is also supported by commercial training courses, consultancy, and other technical services. There are annual international research and user conferences, books [1] and online reference material.

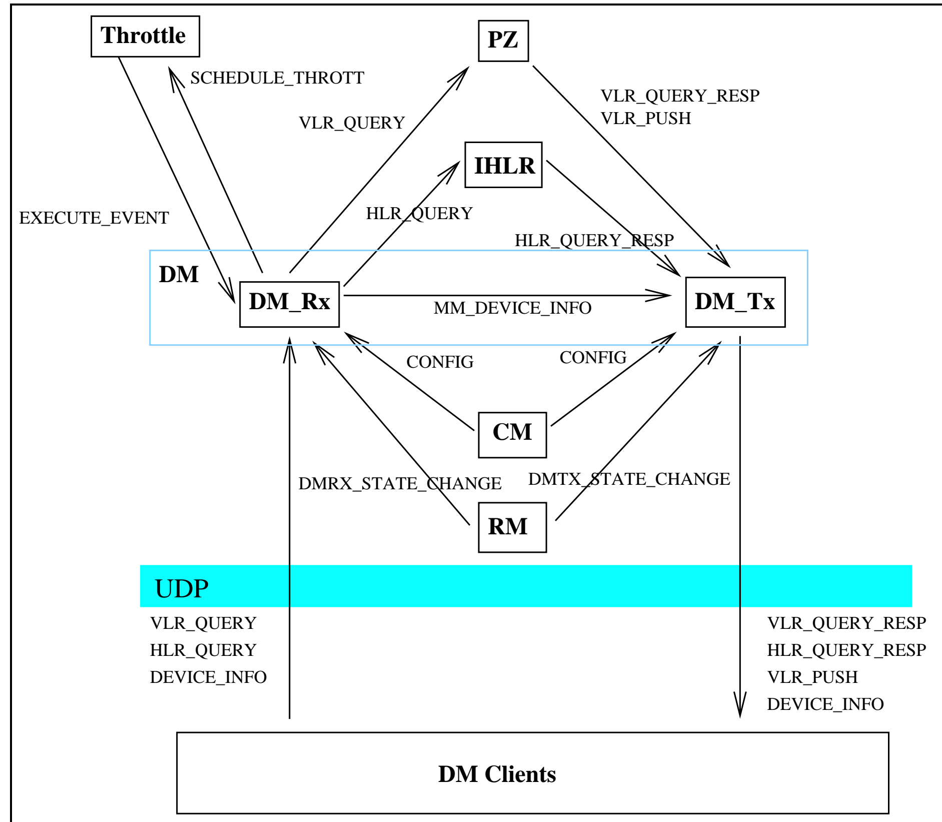

Figure 4: Abstract DM Architecture

4 Basis of Comparison

This section describes the telecom software components that were reengineered as the basis of our comparison between C++ and Erlang.

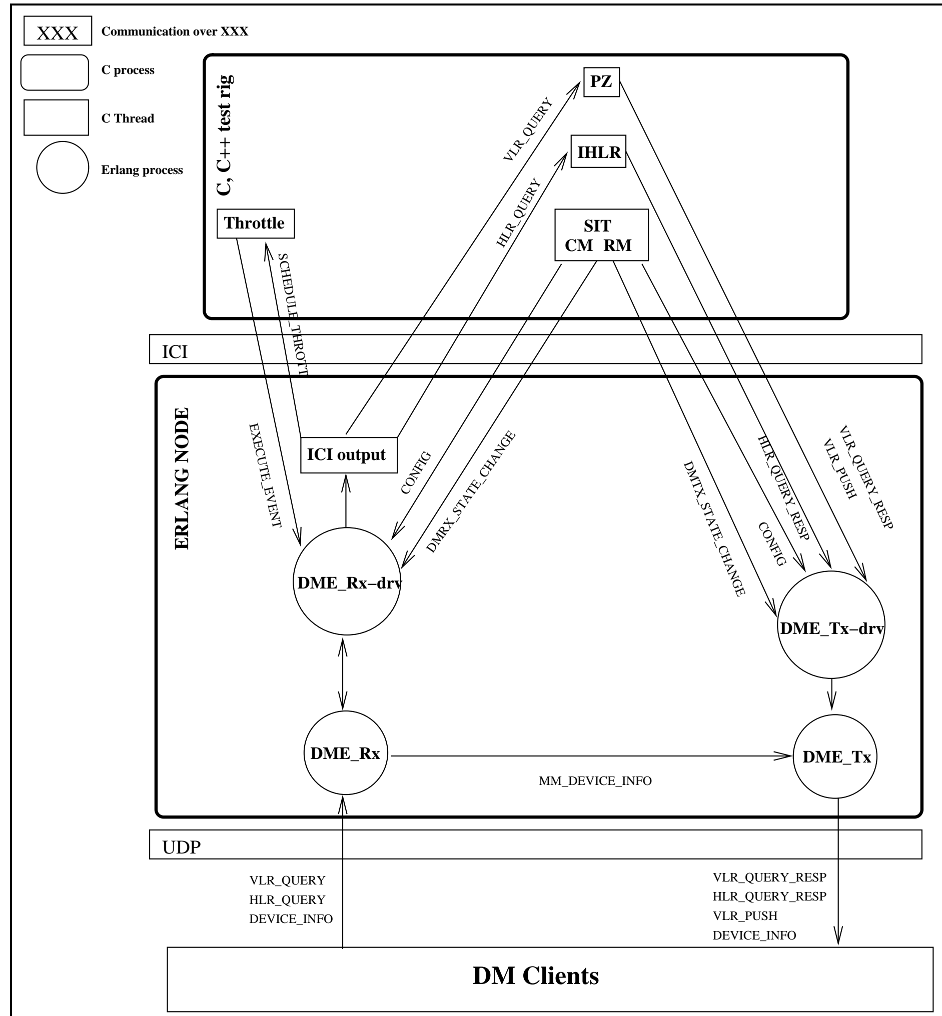

Figure 5: Erlang DM Architecture

4.1 Data Mobility Server (DM) and Platform

The following description of the Data Mobility server (DM) avoids using the proper product names, and some details are made abstract to preserve commercial confidentiality. The DM is a small component of a radio communications

subsystem (RCS) which is responsible for communication between the RCS and data mobility devices. Both the RCS and DM were developed by Motorola as part of an existing product that follows an international standard.

4.1.1 DM Design

The abstract DM architecture is shown in Figure 4, where PZ is a participating zone manager, RM is a resource manager, CM is a configuration manager, and IHLR is an individual home location register. Key aspects of the architecture are as follows. The DM has two main components a receiver (DM_Rx) and a transmitter (DM_Tx). The DM communicates with data mobility devices using UDP, and with five other components of the RCS.

4.1.2 DM Measurement platform

The RCS hardware/operating system platform is SUN/Solaris, and the C/C++ and Erlang DM measurements reported in the following sections have been performed on a 167MHz Sun Ultra 1, with 128Mbytes of RAM, running SunOS 5.8. The C++ compiler is gcc 3.3.3, and the Erlang bytecode compiler is BEAM R10B-6. The BEAM compiler is the most common Erlang platform, although the HiPE compiler that combines native and bytecode code gives better performance for some applications [14].

The architecture of the Erlang/C DM is shown in Figure 5, where DME_Rx and DME_Tx are Erlang receiver/transmitter processes and DME_Rx-drv and DME_Tx-drv are C receiver/transmitter drivers. Key aspects of the architecture are that it combines Unix processes, C threads, and Erlang processes; and that it interoperates with the same C RCS test harness as the C++ DM.

4.2 Dispatch Call Controller (DCC) and Platform

The second component reengineered is a prototype dispatch call system developed at Motorola Labs in Illinois [16]. 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.

The DCC requires the following functionality. It must provide dynamic scalability, i.e. the ability to adapt to use additional resources while the system is running. It must reclaim resources to enable continuous execution, i.e. ensure that once a service instance has terminated, all of its resources are reclaimed. It must be fault tolerant, and in particular provide continued service despite failures. It must meet soft real time performance criteria, i.e. call management mustn't interrupt the call.

4.2.1 DCC Design

The requirements for the DCC model are derived from the technical report by Lillie [16] and from a set of functional requirements [27]. For the purpose of comparison we use 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 [19, 20].

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

Subscribers are simulated by two processes. The first process issues 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 sets up the call and generates simulated voice traffic. The call is initiated by a request from the caller process which waits for the fixed-end to set up the call and route the traffic. A timer is used to model the user terminating the call if it is not commenced within a short period.

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

The fixed-end is simulated in the framework by a service dealing with both hand-offs and call requests. The service uses a database table to associate subscribers with base radio stations. In consequence a hand-off only changes the record of the subscriber who has moved. For calls a service instance is maintained as a handler for the duration of the call. The handler is maintained for a certain time after the call is completed, and is reused should a new call be setup for the same subscriber within a short period. The call handler in the fixed-end also deals with hand-offs during the call.

4.2.2 DCC Measurement 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 100Mb/s Ethernet. The C++ compiler is gcc 3.3.2, and the Erlang bytecode compiler is BEAM R10B-6.

The software test platform is made up of a number of subsystems described below and the overall architecture is shown in Figure 6.

Test Management Responsible for starting and controlling the System Management and Traffic Generator subsystems during the test. The Test Man-

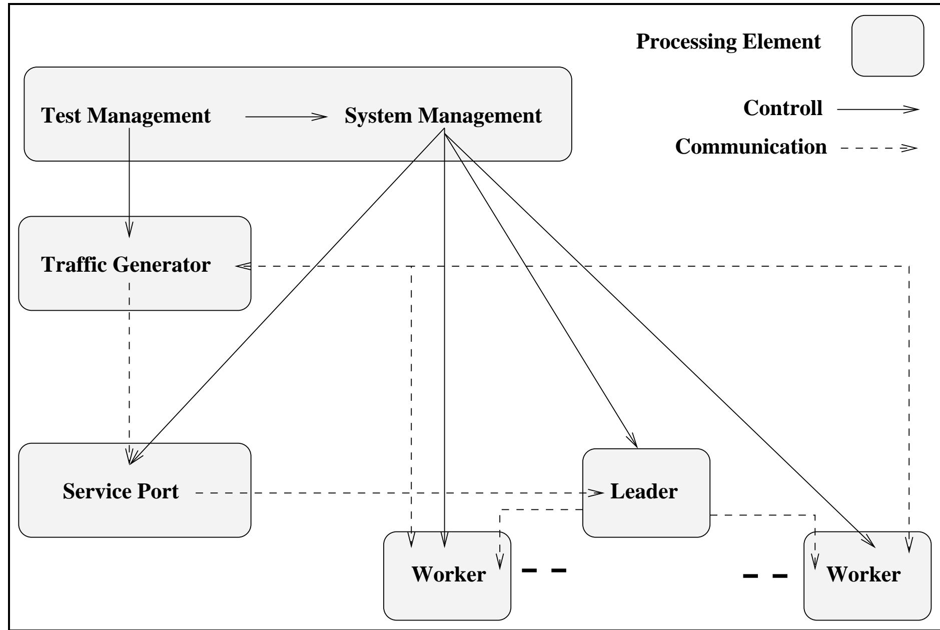

Figure 6: ERLANG DCC Architecture

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

Worker There are 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.

C++ DM	Erlang/C DM	Erlang DM
480	230	940

Table 1: Maximum DM Throughput at 100% QoS

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 is elected using the same protocol.

5 Performance

This section compares the time and space performance of three DM implementations: a C++ implementation, a pure Erlang implementation and an Erlang/C implementation that reuses some C DM libraries. The latter implementation allows the measurement of the costs of interfacing Erlang with foreign code. Some performance measurements for the Erlang DCC are reported in Section 6, but an installation of the C++/CORBA DCC was not available to make systematic performance comparisons.

5.1 Data Mobility

5.1.1 Throughput

The nominal throughput for the data mobility server is less than 1 query/s. Table 1 shows the maximum throughput on the 167MHz Sun Ultra platform specified in Section 4.1.2, for a 100% quality of service(QoS), i.e. handling 100% of all queries. The maximum Erlang DM throughput is approximately double the C++ DM throughput, which is in turn approximately double that of the Erlang/C DM. Given the nominal throughput, we conclude that **both the pure Erlang and interoperating Erlang/C DMs are fast enough for the Data Mobility application**.

5.1.2 Round Trip Times

The round trip times for both types of query, and for a failed query are shown in Figure 7. The Erlang implementations handle failed queries slightly faster than the C++ implementation. **The pure Erlang implementation is approximately three times faster than the C++ implementation**. However, primarily due to the extra layer of communication to the linked in driver, the Erlang/C implementation is 26% slower for successful type 1 queries, and 50% slower for successful type 2 queries.

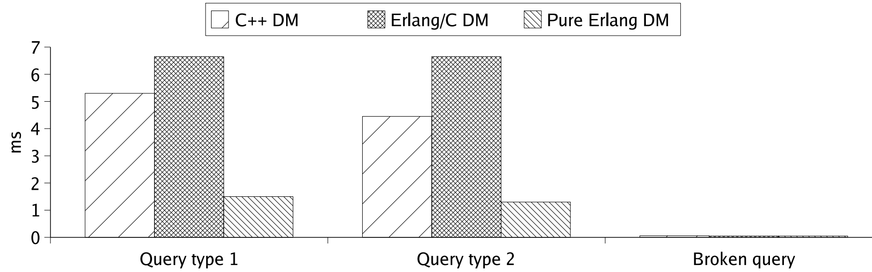

Figure 7: DM Round Trip Times

5.1.3 Space

The maximum residency of the C++ and ERLANG DMs depicted in Figure 8 shows that the ERLANG implementations use significantly more memory. **The Erlang DM uses 150% more memory and the Erlang/C DM uses 170% more**. The dominating space cost (5.2Mb) is the fixed-size ERLANG runtime system (ERTS) which would be a smaller percentage of the total memory cost for applications larger than the 3000-line DM. To put it another way, only the top sections of the ERLANG and ERLANG/C bars, are expected to increase with application size. In the following section we will see the robustness benefits provided by the sophisticated ERTS.

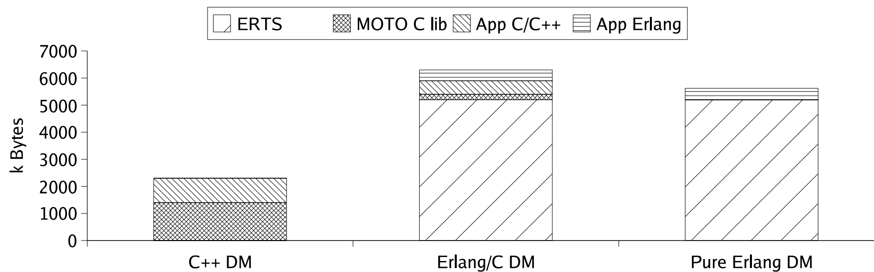

Figure 8: Memory Residency

5.1.4 Performance Discussion

It may initially seem surprising that a bytecode interpreted language like ERLANG outperforms compiled C++. It is straightforward to deduce that for the DM, as with most distributed applications, computation speed does not limit performance, but rather performance is dominated by communication and process management. ERLANG implementations support lightweight processes and are designed to provide fast process management and interprocess communication. In contrast, a sequential language like C or C++ must rely on operating

system process management and interprocess communication. This entails making expensive system calls that switch from user to superuser space, and relying on relatively slow and heavyweight operating system threads. Indeed interprocess communication remains expensive even using shared-memory as in the DM.

6 Robustness

This section investigates the robustness of the DM and DCC applications, covering resilience, availability, and dynamic reconfigurability. Additional measurements of the DCC availability can be found in [20].

6.1 Resilience

Overloading the system should result in a graceful degradation of performance, and the system should recover without human intervention once the load is reduced. Resilience in response to extreme overload is critical in the telecoms and other service-oriented sectors where extremely high service demands occur regularly. For example network traffic peaks immediately after dramatic events like a bombing.

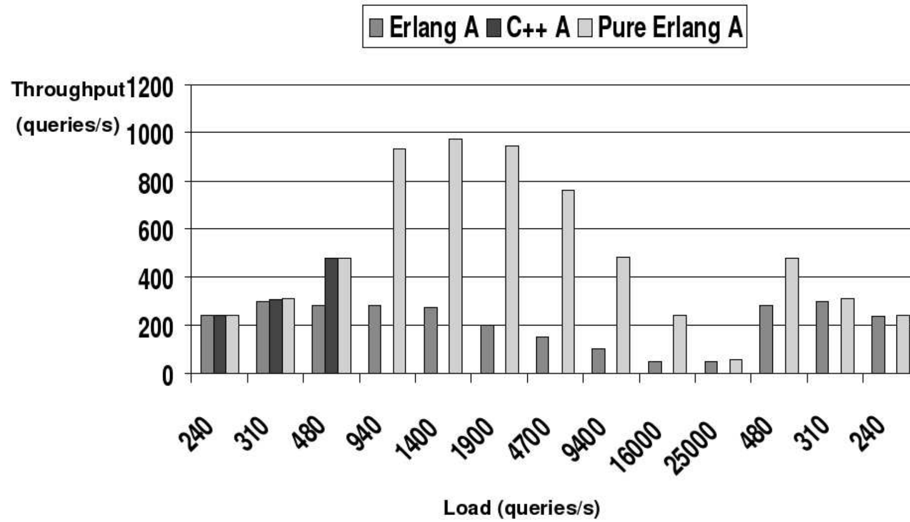

Figure 9: ERLANG/C DM Resilience

6.1.1 DM Resilience

Like most telecoms systems the Data Mobility server controls load by declining new requests. Figure 9 depicts the throughput as the C++, ERLANG/C, and

Erlang DMs are overloaded beyond the nominal load of 1 query/s. The key features are that **although the performance of the Erlang DM degrades, it never completely fails**, e.g. both Erlang DMs handle approximately 50q/s at a load of 25000q/s. Moreover, without human intervention **the Erlang DM recovers after load drops**, whereas the C++ DM has crashed by 920q/s and would require a manual restart.

Of course the C++ DM could be reengineered to decline excess requests, and also to recover. However this would make the program even larger and more complex, as compared with the Erlang program (see Section 7.1). In short, fault tolerance comes more or less for free in Erlang, while it requires considerable extra effort to include and validate it in most other languages including C++. Consequently, applications in these languages are typically far less resilient in practise.

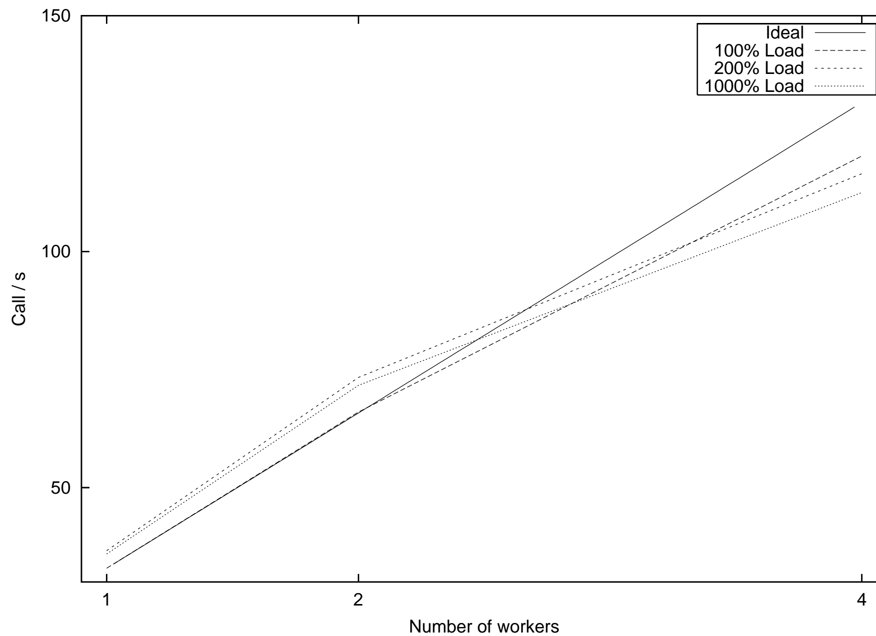

Figure 10: Erlang DCC Resilience

6.1.2 DCC Resilience

For the DCC, an overload is more simultaneous service calls than the system can handle. Like the DM, the DCC controls load by declining requests. DCC resilience is investigated by subjecting configurations of the system to 100%, 200% and 1000% load with a varying number of worker nodes.

The results are reported in Figure 10, and show 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 single worker DCC to 50000% load and the throughput was still 110%. Limitations of the load generation technology prevent measurements of larger systems at such exceptionally high loads.

The C++/CORBA DCC does not contain resilience mechanisms, e.g. to decline requests. In consequence we predict that the C++/CORBA DCC would, like the C++ DM, fail catastrophically when significantly overloaded. However without a usable C++/CORBA DCC we are unable to substantiate this prediction. Like the DM, the C++/CORBA DCC could be be reengineered with resilience mechanisms at the cost of program size and complexity.

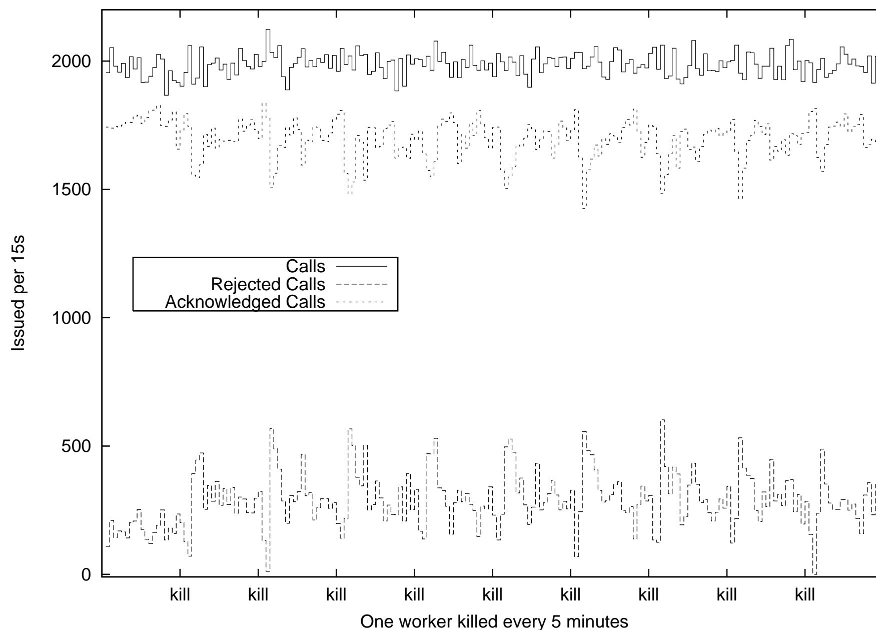

Figure 11: DCC Repeated failures

6.2 Availability

Hardware and software redundancy enables distributed systems to be highly available, that is to continue to function despite hardware or software failures. Ideally the system should continue to function not only in the event of a single component failing, but also when components repeatedly fail and when several components fail simultaneously. One 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 causes other components to fail.

We investigate availability using the DCC primarily as it has multiple processes on multiple nodes. In contrast the DM only comprises four processes in a single node. Even so we have measured the DM with transient errors induced by injecting errors that trigger randomly, and find that the effect on the throughput was too small distinguish from normal fluctuations induced by the varying latency of the network.

To test the DCC for single failures, repeated failures and multiple failures we have subjected a system with 5 worker nodes to the series of tests outlined below. Additional measurements of the DCC availability can be found in [20], including the failure of different node types, including the failure of the leader, and of the leader-elect. In the results of the tests reported in Figures 11, 12, and 14, Acknowledged Calls are the key throughput measure, recording the total number of calls handled by the system. The difference between the Acknowledged Calls and (total) Calls is a quality of service measure, and the initial throughput in the figures represents 100% throughput at the nominal quality of service.

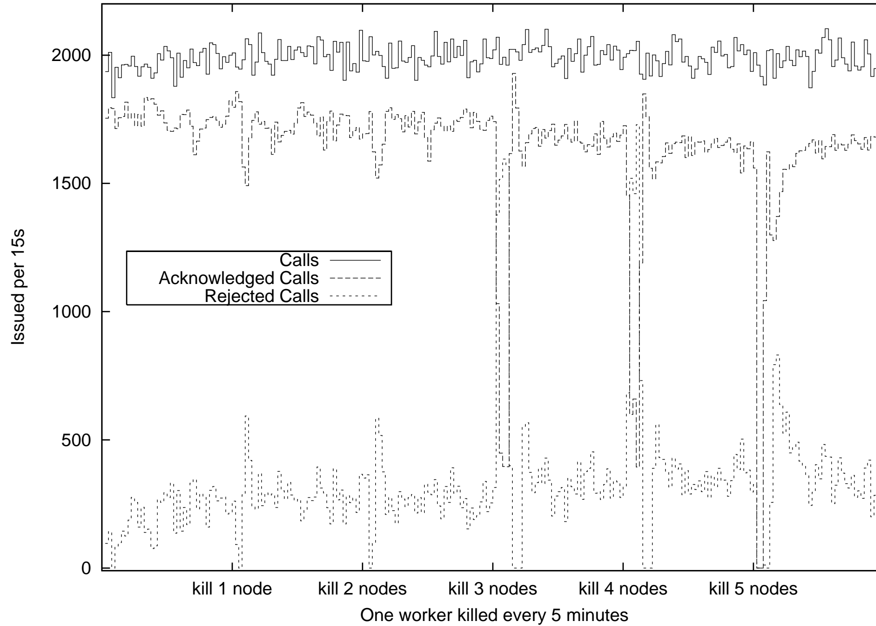

Figure 12: DCC Multiple failures

The availability measurements undertaken are as follows.

- Repeated software and hardware failures. Figure 11 shows that the Erlang DCC remains available despite repeated software failures simulated by destroying the Erlang node on a processor. Figure 14 shows that the

ERLANG DCC remains available despite repeated hardware failures simulated by removing processors. After removing a processor, the throughput is reduced proportional to the number of surviving processors. In summary, the ERLANG DCC **remains available despite repeated hardware and software failures**; and that **performance doesn't degrade with repeated failures** as the throughput after the repeated failures in Figure 11 is the same as at the start.

- Groups of nodes of increasing size are crashed. Figure 12 shows that when more components fail, throughput drops lower and recovery takes longer. However the system recovers to its original throughput within a reasonable time even if all but one of the nodes has failed. In summary, the ERLANG DCC **resists the simultaneous failure of multiple components**.

- Repeated and multiple failures of the service instance processes. Figure 13 shows that the ERLANG DCC **throughput is not significantly reduced when a small percentage of messages crash the service instance**.

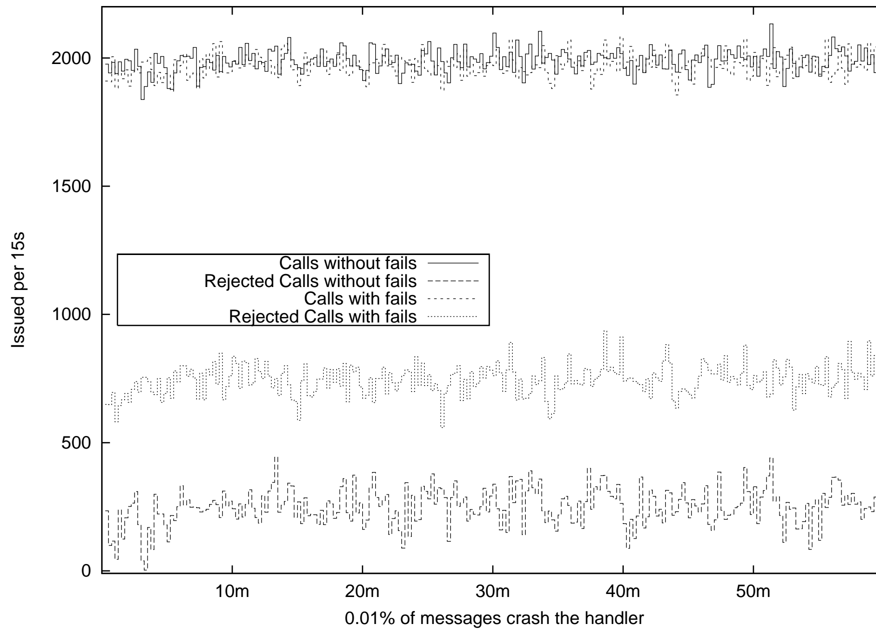

Figure 13: DCC Service Instance Failures

As the C++/CORBA DCC does not contain fault tolerance mechanisms, e.g. to detect and recover from process failures, we predict that the C++/CORBA DCC would fail catastrophically in the event of a hardware or software failure. However without a usable C++/CORBA DCC we are unable to substantiate this prediction. The C++/CORBA DCC could be be reengineered with fault

tolerance mechanisms, or to use a fault-tolerant CORBA, at the cost of program size and complexity.

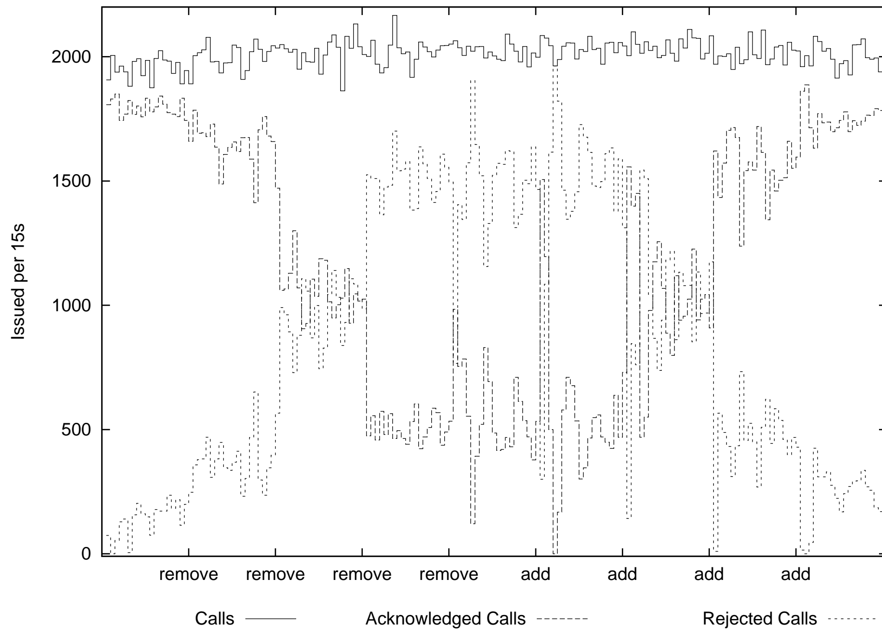

Figure 14: DCC Dynamic Reconfigurability

6.3 Dynamic Reconfigurability

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

The ERLANG DCC behaviour as computational resources are added and removed is evaluated as follows. The system is started with 5 worker nodes and nodes are removed until only the leader node remains and then the worker nodes are added back until the original configuration is obtained. The system has been exposed to a 100% load for 5 worker nodes, using the same notion of load as in the previous section. Figure 14 shows the result of the experiment and illustrates the following properties.

- The second half of the graph shows **near-linear throughput scaling as resources are 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.

Appl.	DM			DCC			
Lang.	C/C++	Erlang	**Total**	C++	IDL	Erlang	**Total**
C++	3101		**3101**	14774	83		**14857**
Erlang/C	247	616	**863**				
Erlang		398	**398**			4882	**4882**

Table 2: DM and DCC Code Sizes (SLOC)

- The first half of the graph shows **near-linear decrease in throughput as resources are 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 cost of adding or removing a processor is small**: the fall in throughput as Erlang nodes are added and deleted is small.

The C++ DCC provides dynamic reconfigurability using CORBA, although systematic measurements are not reported [16].

7 Productivity

This section compares software productivity measures of the C++ and Erlang DM and DCC implementations. The significance of software size is well established: shorter programs are faster to produce [15, 26], and hence programmers working in higher level languages are more productive. The reduced development time crucially reduces time to market for the product. More significantly, given that more than half of programming effort is expended on maintenance, shorter programs are easier to maintain [15].

The metric we use for software size is logical source lines of code (SLOC). There are numerous software complexity metrics, e.g. McCabe's cyclomatic complexity [17], and a good survey is available in [6]. Indeed McCabe's cyclomatic complexity is a Motorola corporate standard but is unavailable for either the DCC or the DM in isolation. SLOC has the advantages of simplicity, relatively wide use, and cross-paradigm applicability. That is SLOC can be used on both the object-oriented C++ and the functional Erlang.

7.1 DM and DCC Code Sizes

The size of the C++ and Erlang DM and DCC are reported in table 2, and the sizes of the DMs are depicted in Figure 15. The pure Erlang DM is 1/7th of the size of the C++ DM, and the Erlang/C DM is 1/3rd of the size of the C++ DM. **Both the pure Erlang DM, and the Erlang/C DCC are less than 1/3rd of the size of the C++ implementations**. These results are consistent with other measurements [30], and with developer folklore in companies like Ericsson, T-Mobile and Nortel.

Part	Lines of Code	Percentage	Modules
Reusable Platform	2994	61%	26
Specific Service	147	3%	1
Testing/Statistics	1741	36%	11
Total	4882	100%	38

Table 3: ERLANG DCC Reusablity

A key productivity issue is reuse, covering both generic services and libraries. The ERLANG/C DM is dramatically smaller than the C++ DM together with the reused RCS libraries: 1/18th of the size (18000 vs 870 SLOC). The library functions are not required by the ERLANG DM. The DCC is an instance of a generic distributed server and Table 3 shows that the majority of the code (61%) is a reusable platform. The table analyses the DCC code sizes into Platform: the reusable generic parts; Testing/Statistics; and Service: the parts specific to the DCC.

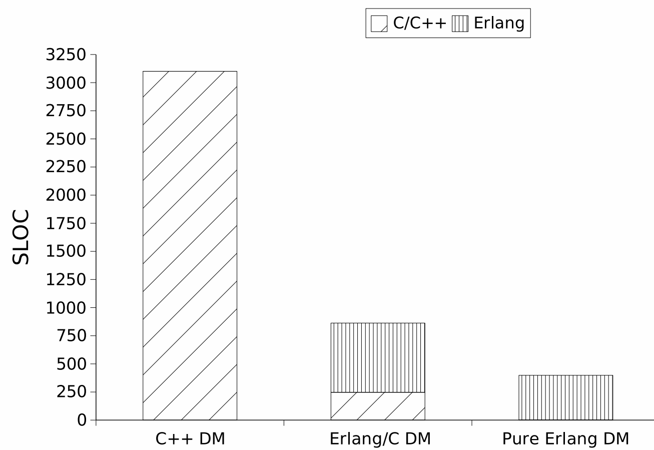

Figure 15: Source Code Sizes

7.2 Reasons for Size Difference

The reasons why ERLANG programs are shorter become apparent from the analysis of the C++ and ERLANG DM code presented in Figure 16 and in Tables 4 and 5. For simplicity we compare the C++ DM only with the pure ERLANG DM, and when interpreting the percentages, the reader should recall that the ERLANG DM is 1/7th of the size of the C++ DM.

- ERLANG's sophisticated fault tolerance mechanisms mean that the programmer can code for the successful case. Hence, **there is far less de-**

Code Type	Erlang Total	Erlang/C Total	Erlang/C Erlang Part	Erlang/C C Part
Application	62.2%	61.8%	69.0%	43.7%
Defensive	0.5%	1.7%	0.0%	6.1%
Communication	15.1%	10.2%	10.9%	8.5%
Memory management	0.0%	3.2%	0.0%	11.3%
Type declarations	4.9%	6.1%	5.2%	8.5%
Defines	5.4%	5.7%	7.0%	2.4%
Includes	2.4%	5.7%	2.4%	13.8%
Process management	9.5%	5.6%	5.5%	5.7%

Table 4: Erlang DM Code Proportions

Code Type	C++ Code	RCS C libraries
Application	19.2%	12.1%
Defensive	25.3%	24.2%
Communication	22.1%	5.6%
Memory management	11.3%	7.1%
Type declarations	11.2%	11.6%
Defines	1.1%	23.6%
Includes	8.1%	8.6%
Process management	1.9%	7.1%

Table 5: C++ DM Code Proportions

fensive code in the Erlang implementations: 0.5% as opposed to 25.3%

- **Erlang's high-level communication greatly reduces communication coding effort** of buffering, marshalling and error-checking: 15.1% as opposed to 22.1%. Figures 2 and 3 give a dramatic comparison of the same communication in Erlang and in C++.

- **Erlang's automatic garbage collection greatly reduces memory management coding effort**: 11.3% as opposed to 0%.

Significantly, much of the code that is omitted from the Erlang implementation is technically challenging, e.g. memory management and defensive code are notoriously hard to get correct and to test. Moreover, Motorola engineers observed that the DM has relatively little defensive code, and that many components contain more than 50% defensive code.

8 Related Work

The impact of the programming language used on the quality and timeliness of software engineering projects has been long established, e.g. [15]. There are numerous comparisons of programming languages, some general e.g. [4], some comparing within a paradigm e.g. object-oriented languages [21] and others comparing paradigms, e.g. scripting versus general purpose languages [26]. Proponents of ERLANG, along with proponents of other functional languages like Clean and Haskell, have participated in these studies. Some of the comparisons incorporate hundreds of languages, but the great majority of benchmark programs are sequential and are necessarily small kernels of larger applications. Rather than small sequential programs, this paper compares two substantial distributed product components, where robustness and configurability are key aspects.

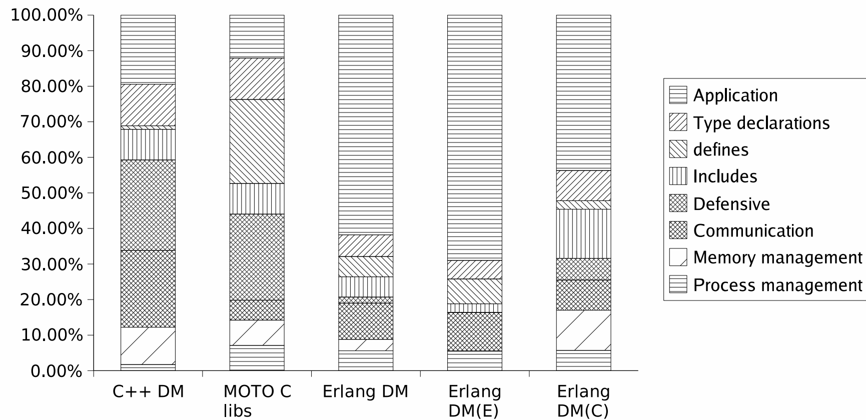

Figure 16: Source Code Breakdown

In contrast to ERLANG, other distributed functional languages like Kali Scheme [3], Facile [8], OZ [10], and Glasgow distributed Haskell (GdH) [25] are all research languages, and very few comparisons with other programming languages have been reported. The comparisons that do exist, e.g. [24], are far smaller scale than those reported here: considering smaller components, and fewer aspects of distributed software engineering.

Comparisons between ERLANG and other language technologies have been reported. Ericsson have undertaken some unpublished comparative studies, although a published study is based on the development of the AXD301 ATM switch which is in excess of 1M SLOC. The study reports that the ERLANG systems have between 4 and 10 times less code than C/C++, Java or PLEX. Moreover, the systems developed in ERLANG and in conventional languages exhibit similar error rates per unit SLOC, and similar SLOC per unit time pro-

grammer productivity. Hence, the ERLANG system shows at least a fourfold gain in productivity and reduction in errors [30]. Our productivity results in Section 7.1 are in close agreement with Wiger's. In contrast to Wiger's general comparison of ERLANG with other technologies, we report a systematic investigation of the six research questions identified in the introduction, and make direct comparisons between corresponding C++ and ERLANG implementations of two components.

In another study Jantsch *et al* compare six languages including ERLANG, for system level description [12]. They report that while ERLANG is suitable for specifying control software, mixed hardware and software systems and simple hardware, it is less suitable for specifying purely-functional software and pure hardware. The study reported here compares ERLANG and C++ for distributed software engineering, rather than for system level description.

9 Conclusion

9.1 Summary

We have investigated high-level distributed language technology for telecoms software by reengineering two telecoms components in ERLANG. Telecoms applications are challenging: requiring the integration of distributed resources under soft real-time constraints and with high availability and configurability requirements. To address the software comparison issues outlined in Section 1.2, two components, one small, and one medium-scale have been engineered. The components have been measured and compared with the existing C++ components. Let us return to the research questions from the introduction, and first consider the potential benefits of a high-level distributed language technology.

Q1 *Can robust, configurable systems be more readily developed?*
Yes, as detailed below.

Resilience The ERLANG DM and DCC both sustain throughput at extreme loads and automatically recover to pre-overload throughput when load drops (Figures 9 and 10). In contrast, as the C++ DM and DCC both lack resilience mechanisms, the C++ DM fails catastrophically when substantially overloaded, and we predict that the DCC would fail similarly.

Availability The ERLANG DCC remains available despite repeated hardware and software failures, and performance doesn't degrade with repeated failures (Figures 11 and 14). The ERLANG DCC resists the simultaneous failure of multiple components (Figure 12). The throughput of the ERLANG DCC is not significantly reduced when a small percentage of messages crash the service instance (Figure 13). In contrast, as the C++/CORBA DCC lacks fault tolerance mechanisms, we predict that it would fail catastrophically in the event of a hardware or software failure.

Dynamic Reconfigurability The Erlang DCC shows near-linear throughput scaling as resources are added and a near-linear decrease in throughput as resources are removed. Moreover, the cost of adding or removing a processor is small (Figure 14). The C++ DCC provides similar dynamic reconfigurability using CORBA.

Q2 *Can productivity and maintainability be improved?*
Yes, using source lines of code (SLOC) as a metric, both the Erlang DM and DCC are less than a third of the size of the C++ counterpart (Table 2). Moreover, much of the Erlang DCC is a reusable generic server (Table 3). The reasons for the reduced programming effort are that coding for the successful case saves 27%, high-level communications save 22%, and automatic memory management saves a further 11% (Tables 4 and 5). These productivity results are consistent with other measurements [30], and with developer folklore.

Secondly, we consider the feasibility of the Erlang high-level distributed language technology for realistic telecoms software development.

Q3 *Can the required distributed functionality be specified?*
Yes, even although low-level distributed coordination aspects are abrogated to the Erlang implementation, the requisite DCC and DM functionality is readily specified.

Q4 *Can acceptable performance be achieved?*
Substantially Yes, Figure 7 shows that the Erlang DMs have acceptable time performance, exceeding the throughput requirements. Indeed the round trip times for the pure Erlang DM are a third of the C++ DM times, and even the Erlang/C round trip times are no more than 50% greater. It may seem surprising that a bytecode interpreted language like Erlang outperforms compiled C++. However in this distributed context where the C++ DM must rely on relatively slow and heavyweight operating system processes and inter-process communication, the Erlang DM uses fast lightweight processes and inter-process communication integral to the language.

In contrast to the excellent time performance, Figure 8 shows that for the DM, as for other small applications, the Erlang memory residency is up to 170% greater due to the large (5Mb) runtime system.

Q5 *What are the costs of interoperating with conventional technology?* Combining the Erlang DMs with the C RCS test harness, and incorporating the C drivers in the Erlang/C DM shows that Erlang components can interoperate with legacy code. As Erlang components are readily made robust, the robustness of a large distributed system can be incrementally improved by (re)engineering critical components in Erlang. Figure 8 shows that the additional space cost of the interoperating Erlang/C DM is small: 15%. However the time penalty for the additional communication with the C driver is high: and the Erlang/C DM round trip

times are 4 times slower (Figure 7) and maximum throughput is a quarter of the pure ERLANG DM (Figure 9).

Q6 *Is the technology practical?*
As far as required by the DCC and DM, ERLANG has proved to be a usable technology. We have shown that ERLANG is available on appropriate hardware/operating system platforms for two typical telecoms products. That is, it is available on the Sun/Solaris RCS product platform for the DM (Section 4.1.2), and on a scalable platform, namely a Beowulf cluster, for the DCC (Section 4.2.2). The technology is well supported with training and consultancy, and many useful components are available in the OTP libraries (Section 3.3.4).

9.2 Discussion

We conclude that high-level distributed languages like ERLANG can deliver the required telecoms functionality and performance. Moreover, such languages offer improved robustness and productivity for distributed telecoms software. The RCS product group have responded very favourably to the robustness and productivity benefits of the ERLANG DMs. An ERLANG DM has been installed alongside the original C++ DM, and the product group are investigating reengineering other parts of the RCS in ERLANG.

Given that ERLANG has significant benefits for the rapid production of robust distributed systems, and was developed at approximately the same time as Java, one might ask why it hasn't been more widely adopted. Although the adoption of a programming language is strongly influenced by social and organisational issues largely beyond the scope of this paper, there are some interesting dissemination issues. One issue is that ERLANG has only become open source recently, unlike Java. Moreover, project managers require strong evidence of potential benefits before adopting a technology, and we hope that systematic studies such as the work presented here will help provide this evidence.

There are also a number of technical issues influencing language choice. Rather than the dominant object-oriented paradigm, ERLANG supports an impure functional programming paradigm. Consequently software engineers require rather specialist training, and do not have access to ubiquitous object-oriented software engineering tools, although specialised tools are available in the OTP. ERLANG was developed in, and even now is primarily employed in a single specialist sector, namely telecoms. Furthermore there is the perception that the performance of a high-level bytecode-interpreted language like ERLANG will compare unfavourably with compiled mid-level languages like C. This perception is, however, mistaken for many distributed contexts where relatively slow sequential execution is more than compensated for by ERLANG's fast process management and interprocess communication, as demonstrated in Section 5. Despite these issues the use of ERLANG, as measured by implementation downloads, is growing exponentially and it is being applied in more and more sectors.

The current work could be extended to explore ERLANG's potential for in-service software upgrades, so-called hot-code loading [1]. In ongoing work we are investigating the impact of a range of language constructs on the engineering of distributed telecoms software, again using the DCC and DM as a basis for comparison. We consider ERLANG, C++ and Glasgow distributed Haskell, and the constructs of interest include the type system, process and communication management, and strict versus lazy evaluation.

References

[1] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in* ERLANG. Prentice Hall, 2nd edition, 1996.

[2] S. Blau, J. Rooth, J. Axell, F. Hellstrand, M. Buhrgard, T. Westin, and G. Wicklund. AXD 301: A new generation ATM switching system. *Computer Networks*, 31(6):559–582, 1999.

[3] H. Cejtin, S. Jagganathan, and R. Kelsey. Higher-order distributed objects. *ACM Trans. On Programming Languages and Systems (TOPLAS)*, 17(1), September 1995.

[4] The Computer Language Shootout Benchmarks. WWW page, July 2006.

[5] O. Dubuisson. *ASN.1 - Communication between heterogeneous systems.* Morgan Kaufmann, 2000.

[6] N.E. Fenton and S.L Pfleeger. *Software Metrics: A Rigorous and Practical Approach.* PWS, 1998.

[7] Al Geist, Adam Beguelin, Jack Dongerra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine.* MIT, 1994.

[8] P. Giacalone, P. Mishra, and S. Prasad. Facile: a symmetric integration of concurrent and functional programming. In *Tapsoft89*, LNCS 352, pages 181–209. Springer-Verlag, 1989.

[9] H. Granbohm and J. Wiklund. GPRS - General Packet Radio Service. *Ericsson Review*, (2), 1999.

[10] S. Haridi, P. Van Roy, and G. Smolka. An overview of the design of Distributed Oz. In *Proceedings of the Second International Symposium on Parallel Symbolic Computation (PASCO '97)*, pages 176–187, Maui, Hawaii, USA, July 1997. ACM Press.

[11] S. Hinde. 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.

[12] Axel Jantsch, Shashi Kumar, Ingo Sander, Bengt Svantesson, Johnny Oberg, Ahmed Hemani, Peeter Ellervee, and Mattias O'Nils. Comparison of six languages for system level descriptions of telecom systems. In *Electronic Chips & System Design Languages*, pages 181–192. Kluwer, 2001.

[13] 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, December 2003.

[14] Erik Johansson, Mikael Pettersson, Konstantinos Sagonas, and Thomas Lindgren. The development of the HiPE system: Design and experience report. *Software Tools for Technology Transfer*, 4(4):421–436, August 2003.

[15] C. Jones. *Programming Productivity*. McGraw-Hill, 1986.

[16] R. Lillie. Implementing dynamic scalability in a distributed processing environment. Technical report, Motorola Labs, Shaumburg, Illinois, 1999.

[17] McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.

[18] MPI-Forum. MPI: A message passing intrface standard. *International Journal of Supercomputer Application*, 8(3–4):165–414, 1994.

[19] J.H. Nyström, P.W. Trinder, and D.J. King. Evaluating distributed functional languages for telecommunications software. In *Proceedings of ACM SIPLAN Erlang Workshop '03*, pages 1–7. ACM, 2003.

[20] J.H. Nyström, P.W. Trinder, and D.J. King. Are high-level languages suitable for robust telecoms software? In *Proceedings of the 24th International Conference, SAFECOMP 2005*, volume LNCS 3688, pages 275–288. Springer-Verlag, 2005.

[21] Object-Oriented Languages: A Comparison. WWW page, July 2006.

[22] Anders Olsen, Ove Færgemand, B. Møller-Pedersen, Rick Reed, and J. R. W. Smith. *Systems Engineering Using SDL-92*. Elsevier, 1997.

[23] D. Pilone and N. Pitman. *UML 2.0 in a Nutshell*. O'Reilly, 2005.

[24] R.F. Pointon, S. Priebe, H-W. Loidl, R. Loogen, and P.W. Trinder. Functional vs Object-Oriented Distributed Languages. In *Eurocast'01*, LNCS 2178, pages 642–656, Canary Islands, Spain, February 2001. Springer-Verlag.

[25] R.F. Pointon, P.W. Trinder, and H-W. Loidl. The Design and Implementation of Glasgow distributed Haskell. In *IFL'00*, LNCS 2011, pages 101–116, Aachen, Germany, September 2000.

[26] L. Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, 2000.

[27] L. Rittle. Distributed Dispatch Architecture Project (Functional Requirements), 1998.

[28] S. Torstendahl. Open Telecom Platform. *Ericsson Review*, (1), 1997.

[29] U Wiger. Industrial-Strength Functional Programming: Experiences with the Ericsson AXD301 Project. In *IFL'00*, Aachen, September 2000. Presentation Only.

[30] U. Wiger. Four-Fold Increase in Productivity and Quality. In *Proceedings of the International Workshop Formal Design of Safety Critical Embedded Systems (FemSYS'01)*, 2001.