Evaluating Distributed Functional Languages for Telecommunications Software

J.H. Nyström P.W. Trinder*
{jann, trinder}@macs.hw.ac.uk
School of Mathematical and Computer Sciences
Heriot-Watt University
EH14 4AS, Edinburgh, Scotland

D.J. King
David.King@motorola.com
UK Software & Systems Engineering Research group of Motorola Labs
Motorola
Basingstoke, England

ABSTRACT

The distributed telecommunications sector not only requires minimal time to market, but also software that is reliable, available, maintainable and scalable. High level programming languages have the potential to reduce development time and improve maintainability due to their compact code size. Moreover reliability is improved by safe type systems and relatively easy verification.

This paper outlines plans and initial results from a joint project between Motorola and Heriot-Watt University that aims to evaluate the suitability of distributed functional languages for constructing telecommunications software. The evaluation will use the ERLANG and Glasgow distributed Haskell(GdH) languages, and be based on the construction of several typical applications. The evaluation will focus on reliability issues like ease of verification, availability issues like fault-tolerance or resilience, as well as whether the languages deliver the required functionalities, like real-time capabilities. The impact of specific languages techniques will also be assessed, including type system, strictness, validation and distributed coordination. The ERLANG and GdH implementations of the applications will be compared with existing C++/CORBA and Java/JINI implementations.

The first application, a Dispatch Call Controller(DCC), has been constructed in ERLANG and measured on a Beowulf cluster. We find that the DCC *scales*, achieving a relative speedup of 14.5 on 16 processors. The DCC is *resilient*, achieving 105% throughput at 200% load and 56% throughput at 9000% load on 16 processors. The DCC is *fault-tolerant*, remaining available despite any one process or processor failure. The DCC has *dynamic adaptability*, remaining available as processors are added or removed.

*This project has in parts been supported by, UK EPSRC Poject No. GR/R 88137

1. CHALLENGES FOR DISTRIBUTED TELECOMS SOFTWARE

The demands of today's telecoms customer/consumer are simply put: they want *low-cost*, *easy to use* systems that are always *reliable* and always *available*. Achieving such high 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. Moreover, from the industry's perspective there is stiff competition, so reducing the time-to-market is also a grand challenge. Further a typical telecoms provider will aspire to reach the 5-nines of 99.999% availability, which equates to a downtime of no more than 5 minutes 15 seconds in a year.

1.1 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 percent of development time. 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 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 new nodes. *Resilience:* The system performance should downgrade gracefully when overloaded. *Dynamic Adaptability:* To ensure high availability the sys-

tem has to be able to adopt dynamically to both software and hardware upgrades.

1.2 Current Technologies

Currently C++/CORBA or Java/JINI are used to construct many distributed telecommunications products. Higher level programming language technologies, like ERLANG or GdH, 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 [4]. Clearly the language technology used must also meet the other functional requirements of telecommunication applications, e.g. real-time requirements, resource reclamation etc.

2. DISTRIBUTED FUNCTIONAL LANGUAGES

Many distributed functional languages have been constructed with a range of models of processes and communication e.g. Kali Scheme [3], Facile [5] and OZ[7]. This project uses the ERLANG and GdH languages.

2.1 Erlang

Several companies already use the ERLANG high-level language [1] and the OTP libraries [17] to construct distributed telecommunications products. Examples include the AXD 301 scalable backbone ATM switch [2], the first implementation of GPRS for standard packet data in GSM systems [6], and the Intelligent Network Service Creation Environment [8]. Features of ERLANG perceived as desirable for implementing reliable and available software include timeouts, exceptions, and a means for monitoring process termination. To allow for nonstop upgrades the system has primitives for on-the-fly code loading, and an explicit notion of time provides support for soft real time applications.

2.2 GdH

GdH is a research language designed to investigate the construction of reliable distributed applications in high-level languages. Haskell [14] is the *de facto* standard non-strict functional language and GdH is part of the Glasgow Haskell Compiler, arguably the most mature Haskell implementation. GdH combines features of two other variants of Haskell, Glasgow parallel Haskell [18] and Concurrent Haskell, with some additional constructs [16].

GdH has the following features. It supports both parallel distributed computation using two classes of thread: stateless threads and stateful I/O threads. PEs are identified so a program can use resources unique to a PE, like files or a GUI interacting with a user. A remote fork primitive creates an I/O thread on a named PE. Some communication and synchronisation is implicit: threads on one PE can share variables with threads on other PEs. I/O threads can explicitly communicate and synchronise using polymorphic semaphores (`MVars`). Higher-level constructs like channels and buffers are constructed by abstracting over distributed `MVars`. Fault tolerance is provided by distributed exception handling, e.g. an exception can be raised on one PE and handled on another.

Table 1: Distr. Telecoms Language Characteristics

Characteristic	C++/ CORBA	Java/ RMI	ERLANG	GdH
Distrib. Coord.				
Comnctn	Exp	Exp/Imp	Exp	Imp/Exp
Synch.	Exp	Exp/Imp	Exp	Imp/Exp
Type Sys.				
Strength	Weak	Strong	Dynam	Strong
Polymphsm	Subtyp. & O/loading	Subtyping		Hin.-Mil. O/load
Validation				
Proof	Hard	Hard	Easy	Easy
Support	Little	Little	EVT	Little
Laziness				
Evaltn	Strict	Strict	Strict	Lazy

2.3 Comparing Erlang and GdH

ERLANG/OTP is a robust industrial language with a proven track record for the construction of large, reliable telecommunications software. The ERLANG/OTP system has commercial support and several unusual features like hot code-loading and soft real-time constructs. In contrast GdH is a research language with little evidence of industrial-strength application development. However, where ERLANG is simple, impure, strict and dynamically typed, GdH is a complete, polymorphically-typed, non-strict functional language. Moreover, GdH has been developed by the participating research group at Heriot-Watt University, and may be adapted to meet the language requirements identified in the project. In short, GdH offers an advanced and adaptable distributed language platform for the project. Table 1 briefly summarises key technical differences between ERLANG, GdH, and the more conventional C++/CORBA and Java/RMI distributed languages. We write 'Exp' for explicit programmer control of some aspect, and 'Imp' for automatic implicit control and 'Dynam' for Dynamically typed.

3. DISTRIBUTED LANGUAGES EVALUATION

We plan to analyse the impact on a cross section of distributed telecommunications applications, both in scale and functionality, of the use of high level distributed programming techniques available in C/C++, Java, ERLANG and GdH. Below are subsections on the language properties that will be investigated for their effect on reliability, availability and maintainability.

3.1 High Level Distributed Coordination

Distributed coordination has many aspects including creating, scheduling and destroying threads, managing communication and synchronisation between threads and placing threads and data on PEs. C++/CORBA has the lowest level coordination, requiring explicit control of many aspects e.g. communication and synchronisation, ERLANG and Java/RMI are at a similar, higher-level of abstraction, e.g. RMI makes thread location transparent. GdH has the highest level with implicit communication of, and synchronisation on, shared immutable state.

High-level distributed coordination potentially reduces de-

velopment time and improves software quality because low level aspects are automatically and correctly managed. The risk, however, is that crucial coordination aspects are no longer under programmer control. Moreover higher level coordination models potentially make it easier to abstract over common distributed coordination paradigms, like client/server.

3.2 Sophisticated Type Systems

Types improve software quality by constraining programs, and a major challenge is for the type system to be adequately expressive. Components of a distributed system must typically be separately typed, often statically, where messages between components are dynamically typed. As indicated in Table 1 the languages in our study have very different type systems: C and C++ are weakly-typed, with C++ having parametric and subtyping polymorphism. ERLANG is dynamically typed. Java is strongly typed with subtyping polymorphism, and GdH is strongly typed with Hindley-Milner and parametric polymorphism. Moreover, GdH statically types all components and messages of a distributed system.

3.3 Validation/Verification

Validation and verification are very important in the distributed telecommunications sector: the primary validation technique is conformance testing which often accounts for a high percentage of development time. The techniques used for large distributed systems differ significantly from those used for small systems. The requirements for small regular system are specified using Message Sequence Charts [10], and designed using SDL diagrams [13]. System code can be generated directly from its SDL diagram, and properties of the system proved using a validator. Large systems are specified and designed carefully but often evolve during and after the initial implementation and often there is no accurate specification of the system [9].

Declarative languages verify properties applying equational reasoning techniques to the application code. Support tools are available, e.g. the Erlang Verification Tool (EVT) [4] which also includes outline proofs for generic distributed paradigms, like client-server [9]. Potentially these techniques can be applied to larger systems where no specification exits.

3.4 Availability Support

Availability is a crucial quality for many applications areas including telecommunications. Programming languages support availability with capabilities for fault tolerance and hot-loading of upgrades into running systems. The languages in our study have varying levels of support for availability: C++ has almost no explicit constructs, where the other languages all support exceptions, and ERLANG has the most sophisticated additional support with watcher processes, soft real-time timeouts and hot-loading of system upgrades.

3.5 Strictness vs Non-Strictness

In a distributed context non-strictness, or laziness, has potential benefits for communication and for fault tolerance. Under a lazy regime only the required data need be communicated at the cost of additional messages to request the data [15]. Lazy languages also have mechanisms for encapsulating computations that may be useful for restarting computations in the event of a fault.

4. HIGH-LEVEL TELECOMS APPLICATION DEVELOPMENT

To evaluate the properties mentioned in the previous section, as well as investigating the appropriateness of high-level language techniques for distributed telecommunication applications, we plan to develop three medium-scale advanced telecommunications applications in ERLANG and GdH. Comparison is facilitated because the applications have counterparts developed in existing technologies. Three possible candidates are briefly presented in the following subsections and the first preliminary results from the first application implemented in ERLANG, to whit DCC, is described in the next section.

4.1 Dispatch Call Controller (DCC)

The first application is relatively simple: group-call, or *dispatch* call, processing is a prevalent feature of many wireless communication systems (W-LANs). 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 [12].

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.

4.2 A Location-Aware Service Provider (LASP)

The second application detects the location of a person with a portable device (PDA, phone, badge), and provides at their location on a LAN the services described by their profile [11]. A simple example is a person with workstations in both office and laboratory, when they enter the vicinity of either office or laboratory their workstation screen is unlocked and email is downloaded. The range of services is extensible, being provided by a service trader.

Figure 1 is a dataflow diagram of the application. Users record the services they require as a profile on the Trader, and the location of the user's mobile device is detected by wireless-to-LAN access points (AP) on the LAN. When the mobile device detects an AP it sends a message (1) to the *Mobile Proxy* identifying the user, the AP and providing a security key. The Mobile proxy then sends a message (2) to the *Trader* requesting the user's services. The Trader responds with a message (3) containing a list of services available at the location identified by the AP. The mobile proxy then sends a message (4) to the *Locker* requesting the list of services, authorised by the user's key. If the key is correct, the locker instigates the provision of the services (5).

The existing LASP implementation is a Java, C++ and CORBA research prototype. We will construct the mobile

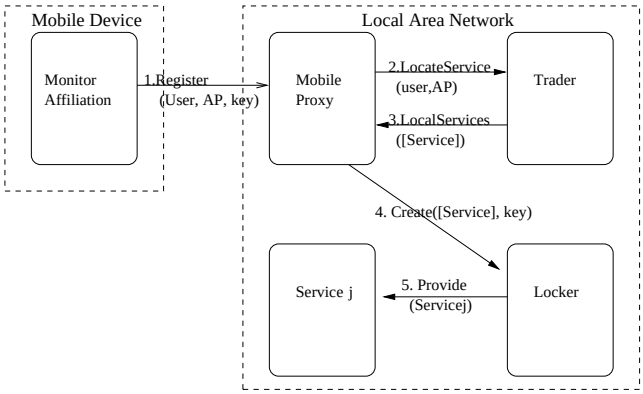

Figure 1: LASP Architecture

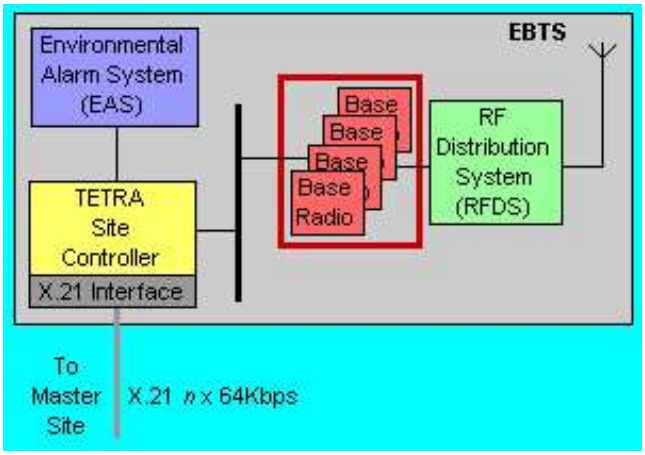

Figure 2: EBTS Site Components

proxy and locker in both ERLANG and GdH, but use an existing ORB, e.g. The Ace ORB (TAO) from Washington University.

4.3 Base Radio Controller (BRC)

The final software application is to re-implement a large part of the Base Radio Controller of the *Dimetra* system (Digital Mobile Enhanced Trunked Radio) which is Motorola's brand name for its 100 percent TETRA (TErrestrial Trunked RAdio) compliant portfolio of networks and terminals. The Dimetra system is a sophisticated trunked radio system which will be used world-wide by emergency services. It has all sorts of communication features including: teleservices – such as group call, emergency call, private calls; telephone interconnection – for communication with ordinary telephones; transmission security with encryption; besides others. To preserve confidentiality, we cannot reveal here its precise functionality, underlying technologies, or implementation characteristics.

Enhanced Base Transceiver System (EBTS) are components of a Dimetra system. An EBTS site (see Figure 2) has four major components: Base Radio (BR); Site Controller (SC); Radio Frequency Distribution System (RFDS); and Environment Alarm System (EAS). For our purposes, we are only interested in the Base Radio Controller which is part of the BR. The Base Radio Controller (BRC) interfaces between the SC and RFDS. The BRC has subsystems that include handling routing, encryption, processing packet data, besides others. BRC software can be tested by setting up a workstation to behave like the BRC, and passing test signals to and from the workstation via Ethernet.

The BRC subsystems provide an enormous range of facilities, many of which are similar and raise no new technical issues. Due to the size of the BRC, we will implement only 40% of its functionality in ERLANG, but the subset will include examples of each class of facility from every subsystem thereby addressing all of the technical challenges.

5. INITIAL EXPERIMENTS

This section describes experiments into: *scalability*, *resilience*, *fault-tolerance* and *dynamic adaptability*, of the DCC application outlined in Section 4.1. First we present the experimental apparatus, and then are listed for each experiment the purpose, method and results.

5.1 Experimental Apparatus

The experiments have been conducted using a simple service that performs non-communicating computation and terminates after 1 second, and each processing element may execute two concurrent service instances. Calls received when the PE is already executing the maximum number of instances, are rejected. This simple service is intended to as act as a baseline for future experiments with more elaborate services such as full Dispatch call processing.

For comparison a baseline implementation, that is non--distributed and without fault-tolerance, of the simple service running on one processing element has been constructed. This can only handle two service calls at any one time, so for comparison with the distributed implementation with more processing elements the values are extrapolated by multiplying the throughput of the baseline implementation with the number of nodes.

The system may have between 1 and 16 service nodes running on separate processing elements and 1 processing element each dedicated to the traffic generator and system interface.

The system is run on a 32-node Beowulf cluster, where each node consists of a Pentium-III 530 MHz processor with 128 Mbytes of RAM, interconnected by a conventional 100Mbit ethernet.

5.2 Scalability

In telecommunication applications distribution is often used to increase performance, and hence throughput should scale as additional processing element are added.

This is investigated by comparing the number of service calls handled per time unit at 100% load with a varying number of processing nodes. The service load is provided by traffic generators simulating service calls from users of the services, where the generator will generate the maximal number of concurrent services instances per processing element times the number of processing elements each second (since each service instances will take one second to terminate).

The result of the experiment is shown in Figure 3, and Figure 4, where the Figure 3 presents the speedup in actual calls per second where the units at the left y-axis are the calls per second being the theoretical upper limit that one

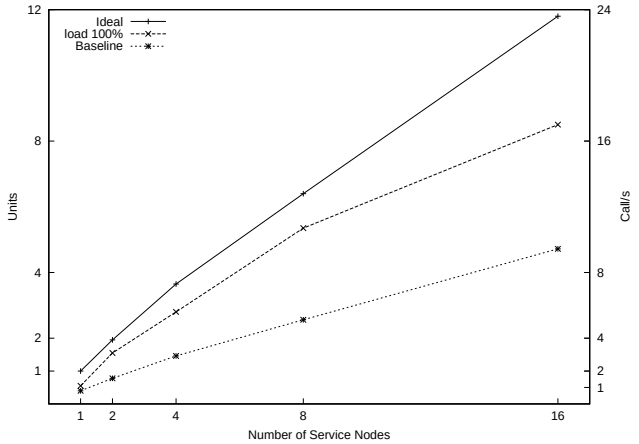

Figure 3: Absolute Speedup

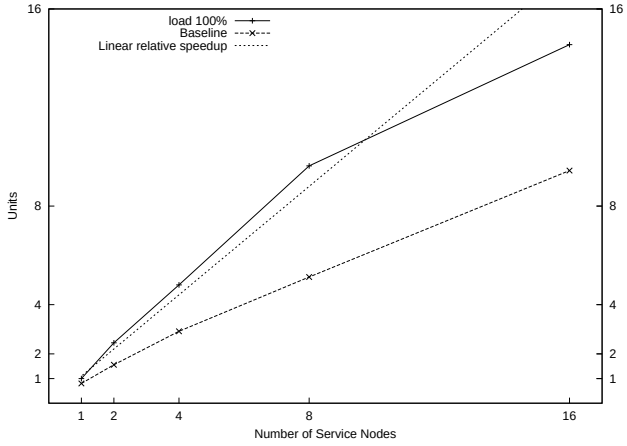

Figure 4: Relative Speedup

Service Node can execute, assuming now overhead for communication and distribution. The second Figure 4 shows relative speedup, the units on the y-axis being the number of service calls handled by one Service Node. Below is described what each individual curve in the figures mean:

Ideal The actual number of calls generated, this being the optimal number of calls serviced. Since the traffic generation is imperfect due to the small time intervals between calls when the number of processing elements is large, the actual number of calls will be smaller than the specified

Load 100% The number of calls handled by the service at 100% load.

Baseline The number of calls handled by the baseline implementation with the values extrapolated as described above.

Linear relative speedup The results we would get if we had linear speedup with increasing Service Nodes.

These results clearly show a reasonable speedup of 14.55 times with 16 Service Nodes. Somewhat counter intuitively the baseline implementation has lower throughput than the distributed fault-tolerant even with 1 processing element, this is because the overhead of distribution and fault-tolerance is less than the gain by the implicit buffering in distributed implementation when dealing with non-regular service calls intervals.

5.3 Resilience

Overloading the system should only result in a graceful degradation of performance. By overload is meant more simultaneous service calls than the system can handle.

This investigated subjecting the system to 100%, 200% and in the excess of 9000% load with a varying number of processing nodes. The same traffic generator as in the scalability experiment is used.

The results of the experiment is shown in Figure 5 and Figure 6, where the first figure show the throughput at different loads and the second shows the results where the curves have been normalised by dividing the number of calls per second by the number of Service Nodes and the baseline implementation at 100% load is assumed as 100% efficiency.

The non-regular service call intervals means that at 100% load there will be times when one or more of the Service Nodes only execute one service instance or is idle, as a result throughput increases as the load increases past 100% since the higher load makes it less likely that Service Nodes are not fully occupied.

The trend that throughput increases as the load increases is broken when the combined communication of the caused by service calls and distribution administration causes congestion in the 100Mbit Ethernet connecting the processing elements. In the experiments network congestion is reached first when we have more than 300 calls per second in a system with 16 Service Nodes, resulting a noticeable performance drop where for the first time the performance is actually worse than that of the baseline implementation.

5.4 Fault-Tolerance

In telecommunications systems it is important to have continuous service in the presence of software and hardware

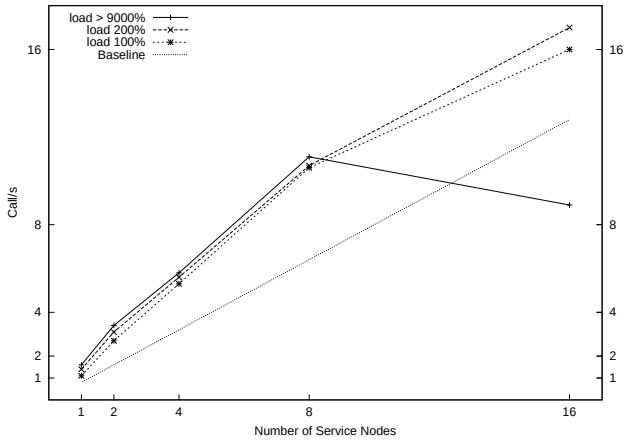

Figure 5: Resilience

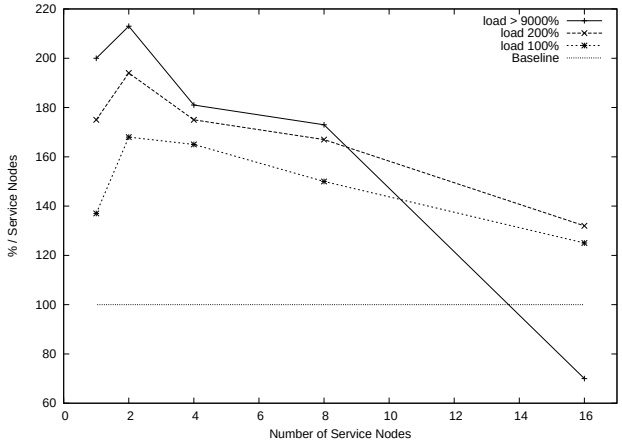

Figure 6: Resilience normalised

faults. The in ERLANG implemented distribution framework fault-tolerance has two aspects: first that all the nodes[1] and processes making up the framework are, when possible, restarted on failure; secondly that no service call received, whether started or not, is lost due to a fault in the system. First aspect is henceforth referred to as *structural fault-tolerance* and then second *transparent fault-tolerance*.

To evaluate system performance in the presence of software and hardware faults, faults are injected, terminating processes, groups of processes and whole nodes. The existing ERLANG primitives for controlling processes and nodes are used to inject the faults, and the behaviour is compared to an undisturbed system. Sofar only *structural fault-tolerance* have been tested for failure of any one node or process. To test for *transparent fault-tolerance* convincingly would involve repeated failures in long runs at different configurations and an is much time consuming, it has only been determined that *transparent fault-tolerance* has not been violated during the experiments to test *structural fault-tolerance*.

No comprehensive evaluation of performance in the presence of failures has been done, but the experience from the *structural fault-tolerance* testing indicates that the performance degradation during fault-handling is acceptable.

5.5 Dynamic Adaptability

To ensure high availability the system has to be able to adopt dynamically to both software and hardware upgrades as well as hardware downgrades. To investigate this aspect the system is subjected to: software upgrades; added or removed nodes; and processes handling individual service calls have their tasks moved to processes on other nodes.

Through experiments it has been verified that one or several nodes can be added or removed from the running system while maintaining *structural fault-tolerance*. No comprehensive evaluation of the performance effects has yet been made, but the initial measurements indicates that they are acceptable. It should be noted that all of the system resources freed by removing a processing element are automatically reclaimed.

6. CONCLUSIONS AND FUTURE WORK

This paper has outlined the plans for, and the initial results from, a joint project between Motorola and Heriot-Watt to evaluate the ERLANG and GdH distributed functional languages for telecommunications software. The evaluation will be based on the construction of several typical applications and will assess both whether the technologies deliver the required functionalities, and the impact of specific language features.

The first application, a Dispatch Call Controller, has been constructed in ERLANG and measured, as follows:

Scalability it achieves a relative speedup of 14.55 times on 16 processing elements.

Resilience it achieves 105% throughput at 200% load and 56% throughput at 9000% load, relative to the throughput at 100%, with 16 processing elements.

Fault-Tolerance the system remains available despite any one process or processing elements failing.

[1]If an ERLANG-node fails, all processes on that node also fail.

Dynamic Adaptability the system remains available when processing elements being added or removed.

We conclude that for distributed telecommunications server applications like DCC ERLANG meets the scalability, resilience, fault-tolerance and dynamic adaptability requirements.

6.1 Future Work

We plan to complete our experiments with the DCC, undertaking performance measurements of the effects of fault-handling and dynamic adaptability.

The set of experiments will be augmented with measurements of the performance impact of fault-tolerance and dynamic adaptability. On-the-fly code loading will also be demonstrated, with measurements of performance effects. In addition more comprehensive and large-scale testing must be made with the DCC to ensure confidence in the system.

Once the complete experiments have been carried out on the ERLANG implementation of the DCC, a GdH implementation will be constructed and measured.

A second nontrivial applications will be implemented: Location-Aware Service Provider, that detects the location of a person with a portable device and provides at their location on a LAN the services described by their profile [11]. This second implementation is made in order to show that high-level technologies facilitate the construction of a generic software platform, that can be instantiated to provide different, but similar distributed telecommunication services, aiding rapid and correct system development.

As a third application to be implemented will be the Base Radio Controller, a re-implementation large part of Motorola's *Dimetra* system in ERLANG. This implementation will be by far the largest and, thereby demonstrating that the technologies used scale to realistic telecommunication applications.

Based on these applications the impacts of the language features outlined in Section 3 will be assessed, in comparison to the existing C++/CORBA and Java/JINI implementations.

7. 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), Sept. 1995.

[4] L. Fredlund, D. Gurov, and T. Noll. Semi-Automated Verification of Erlang Code. In *Proceedings of the* 16^{th} *International Conference on Automated Software Engineering 2001 (ASE'01)*, pages 319–323. IEEE Computer Society Press, 2001.

[5] 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.

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

[7] S. Haridi, P. Van Roy, and G. Smolka. An Overview of the Design of Distributed Oz. In *2nd Intl. Symposium on Parallel Symbolic Computation (PASCO 97)*, New York, USA, 1997.

[8] S. Hinde. Use of ERLANG/OTP as a Service Creation Tool for IN Services. In *Proceedings of the* 6^{th} *International* ERLANG*/OTP Users Conference (EUC'00)*. Ericsson Utvecklings AB, 2000.

[9] F. Huch. Verification of Erlang Programs using Abstract interpretation and Model Checking. *SIGPLAN Notices*, 34(9), 1999.

[10] International Telecommunications Union. *Message Sequence Charts*, 1996. ITU-T Recommendation Z.120.

[11] S. Landis. Investigation into location-aware wireless applications using distributed object technologies. Technical report, Motorola Labs, Shaumburg, Illinois, 1999.

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

[13] A. Olsen, O. Rærgemand, B. Møller-Pedersen, R. Reed, and J. R. W. Smith. *Systems Engineering Using SDL-92*. Elsevier, 1997.

[14] J. Peterson, K. Hammond, et al. *Report on the Programming Language Haskell (Version 1.4)*, Apr. 1997.

[15] R. Pointon, S. Priebe, H.-W. Loidl, R. Loogen, and P. Trinder. Functional vs Object-Oriented Distributed Languages. In *Eurocast'01*, Canary Islands, Spain, Feb. 2001. Springer-Verlag. To appear.

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

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

[18] P. Trinder, K. Hammond, H.-W. Loidl, and S. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, Jan. 1998.