Evaluating High-Level Distributed Language Constructs

Abstract

The paper investigates the impact of high level distributed programming language constructs on the engineering of realistic software components. Based on reengineering two non-trivial telecoms components, we compare two high-level distributed functional languages, ERLANG and GdH, with conventional distributed technologies C++/CORBA and C++/UDP.

We investigate several aspects of high-level distributed languages including the impact on code size of high-level constructs. We identify three language constructs that primarily contribute to the reduction in application size and quantify their impact. We provide the first evidence based on analysis of a substantial system to support the widely-held supposition that high-level constructs reduce programming effort associated with specifying distributed coordination. We investigate whether a language with sophisticated high-level fault tolerance can produce suitably robust components, and both measure and analyse the additional programming effort to introduce robustness. Finally, we investigate some implications of a range of type systems for engineering distributed software.

General Terms Distributed Functional Languages, Industrial Applications

1. Introduction

There has been sustained interest in constructing high-level distributed languages, e.g. Kali Scheme [3], Facile [6], OZ [9], ERLANG [1] and Glasgow distributed Haskell (GdH) [22]. Like other language designers, distributed language designers propose new constructs and demonstrate them on small exemplars. However a realistic assessment of a construct, and especially one intended for large scale distribution, must be based on substantial realistic exemplars.

This paper investigates the impact of high level distributed language constructs on the engineering of realistic software. Unlike the numerous programming language comparisons based on sequential kernel benchmarks, e.g. [4, 19], we base our evaluation on two distributed components, where distributed coordination, and especially robustness, is a key aspect. Moreover, rather than being benchmark kernels, the components are substantial and form part of telecom products.

Earlier work has compared ERLANG and C++/CORBA for distributed telecoms software, focusing on distributed software requirements: robustness, productivity, performance, functionality, interoperability and practicality [17]. The results demonstrate that the high level ERLANG components meet the functional requirements, and hence can be reasonably compared with the existing C++ based implementations.

[Copyright notice will appear here once 'preprint' option is removed.]

Here we focus on the impact of high-level programming language constructs on the construction of realistic distributed software. The high-level languages are ERLANG, an industrial-strength language with high-level distributed coordination and advanced fault tolerance mechanisms [1], and Glasgow distributed Haskell (GdH) a research language with very high-level distributed coordination [22]. Our research strategy is to reengineer two telecoms components in ERLANG and GdH and make comparative measurements with the existing implementations using conventional distributed technology, i.e. C++ combined with CORBA and with communication libraries. The first component is a medium-scale (15K line) Dispatch Call Controller (DCC) [15] (Section 4.2). The second is a smaller (3K line) Data Mobility (DM) component that is closely integrated with five other components of a base radio network (Section 4.3).

We investigate the following research questions.

Q1 Do high-level distributed language constructs reduce application size? Software size is crucial as shorter programs are faster to produce and easier to maintain. Our investigation compares the sizes and functionalities of implementations of two telecoms components in the high-level GdH and ERLANG with conventional implementations in C++/CORBA and C++/UDP (Section 5).

Q2 What high-level distributed language constructs impact application size? Our investigation quantifies the impact of language constructs such as garbage collection, high-level communication and advanced fault tolerance on code size (Section 6).

Q3 What is the impact of high-level coordination? In theory a language with high-level coordination reduces the programming effort of specifying coordination. We report what we believe is the first investigation of this hypothesis in the context of realistic distributed components, and cover both ERLANG and GdH (Section 7).

Q4 What is the impact of sophisticated fault tolerance constructs, and what are the costs of introducing robustness? We investigate how readily ERLANG, a language with advanced high-level fault tolerance, can produce suitably robust components. We further measure the cost of introducing robustness in the context of realistic distributed components by measuring both the DM and two versions of the DCC, one with, and one without, fault tolerance (Section 8).

Q5 What are some of the impacts of the type systems on distributed software engineering? The languages in our study have very different type systems: ERLANG is dynamically typed, C++ is weakly typed with parametric and subtyping polymorphism, and GdH is strongly typed with Hindley-Milner and parametric polymorphism. We investigate the some of the implications of these type systems on the construction of a substantial distributed component, measuring the numbers of

type declarations and the frequency of dynamic type checking (Section 9).

2. Related Work

Popular programming language comparisons are inappropriate for distributed languages. There are few distributed language comparisons and they typically compare performance rather than language, e.g. the Hartstone distributed benchmark measures real-time performance [14]. There are numerous language comparisons, some general e.g. [4], some comparing within a paradigm e.g. object-oriented languages [19] and others comparing paradigms, e.g. scripting versus general purpose languages [23]. Some of the comparisons incorporate hundreds of languages, and some distributed languages, e.g. ERLANG, have contributed to the comparisons. However the great majority of benchmark programs are sequential and are necessarily small kernels of larger applications, and hence not a good basis for comparing languages intended for engineering large-scale distributed systems. In contrast this paper analyses the impacts of distributed language constructs, e.g. fault tolerance, on the engineering of two substantial distributed product components.

Evaluating distributed functional languages (DFLs) with substantial real applications is challenging. Most DFLs like Kali Scheme [3], Facile [6], OZ [9], and GdH [22] are research languages and are demonstrated on relatively small 'virtual' applications. Virtual applications are only used for measurement, and not for a practical purpose. Our study is unusual in comparing, *inter alia*, ERLANG and GdH using substantial real distributed product components.

Comparing distributed languages, and especially DFLs, is challenging. DFLs require a sophisticated implementation to manage distributed coordination, and as research languages most DFL implementations are relatively immature. For example a language may only be available as a prototype on a specific hardware/operating system platform. Some small scale comparisons have been undertaken, e.g. [21] compares Eden, GdH and Java/RMI using small kernels. We use larger components and compare four distributed technologies: the ERLANG and GdH DFLs, and C++ with CORBA and UDP.

Other studies have compared ERLANG with other distributed language technologies. Ericsson have undertaken some unpublished comparative studies. A published study addresses research question Q1 in this paper, i.e. the reduction in application size. The study is based on the development of the AXD301 ATM switch with more than 1M source lines of code (SLOC) and reports that the ERLANG systems have between 4 and 10 times less code than C/C++, Java or PLEX [28]. This is in agreement with the results we report in Section 5. Our study goes beyond these earlier studies by making a systematic investigation of the impact of specific distributed language constructs as outlined by the research questions in the previous section.

Earlier work has thoroughly investigated the fault tolerance of the ERLANG DCC [18], and compared ERLANG, but not GdH, with C++/CORBA and C++/UDP for distributed software engineering [17]. The latter work compares the robustness, productivity, performance, functionality, interoperability and practicality of the technologies. The results demonstrate that the ERLANG components meet the corresponding functional requirements, and hence are an appropriate basis for the language analysis reported in sections 5 – 9.

3. Distributed Language Technologies

Distributed coordination can be specified at a range of levels of abstraction. At the lowest level the programmer explicitly places computations and resources at named locations and arranges communication and synchronisation between them. Such low-level coordination, e.g. `send`/`receive` or `wait`/`signal` can be viewed as equally harmful for coordination as `goto` is for algorithms [7]. Higher level distribution technologies reduce the programming effort to specify coordination, and this section briefly outlines the distributed technologies used in our study.

C++ is a sequential language, and like many other sequential languages is commonly combined with distribution technologies to construct distributed systems. The two distribution technologies used for the telecoms components measured here are CORBA and the UDP and ICI communication libraries.

3.1 Low-Level Distribution: UDP and ICI

Constructing a distributed system using a low-level communication library typically entails explicitly spawning operating system processes and managing communication between them. Often data must be explicitly marshalled and unmarshalled. There are numerous libraries at different levels of abstraction. Both the UDP and ICI libraries used in our study are low-level. The User Datagram Protocol (UDP) is a core internet protocol, and enables components to send short messages, or datagrams, over sockets. UDP is relatively fast as, unlike TCP, it does not provide reliable or ordered datagram delivery.

3.2 Mid-Level Distribution: CORBA

CORBA, Common Object Request Broker Architecture, wraps a sequential component into an object containing information about the capabilities of the component and how to call it. The wrapped objects can be called from other programs or CORBA objects across a network. CORBA uses an interface definition language (IDL) to specify the interfaces that objects will present to the world. CORBA then specifies a mapping from IDL to a specific implementation language like C++ or Java. CORBA provides mid-level distributed coordination by providing a language and platform neutral remote procedure call. In addition CORBA defines common services such as transactions and security.

3.3 Higher-Level Functional Distribution

A number of distributed functional languages have been constructed with a range of models of distributed coordination, as outlined in the previous section. With the exception of ERLANG, almost all are research languages used to investigate high-level distributed coordination integrated with the language. The following sections briefly outline the DFLs used in our study.

3.4 High-Level Distribution: ERLANG

ERLANG is a DFL originally developed in Ericsson for constructing highly reliable telecom systems [1]. The language has integrated distribution, including first-class processes, advanced 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 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.

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 sectors, e.g. banking. Examples include the first implementation of GPRS for standard packet data in GSM

systems [8], and the Intelligent Network Service Creation Environment [10]. 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 [11, p.6], 300K lines of mostly-reused C and 8K lines of Java, developed by a team peaking at 50 software engineers [27].

3.4.1 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.

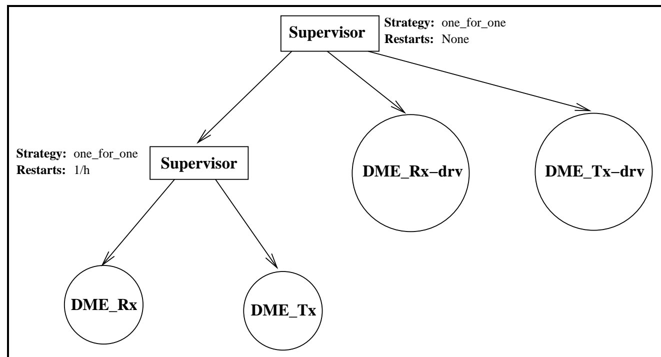

Figure 1. Erlang/C DM Supervision Tree

The monitoring of one process by another is sufficiently common that it is encapsulated by the *supervisor behaviour* [11]. 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 4.3. 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

3.4.2 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++ with ICI. 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 32-37 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.

```
1void 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
12 // sends a Device Info message to Q1 and Q2 clients
13 IciMsg ici_msg_object( MM_DEVICE_INFO_OPC,
                          ICI_DMTX_TASK_ID, msg_size);
15 // Retrieve ICI message buffer pointer
16 msg_buf_ptr = ici_msg_object.getIciMsgBufPtr();
17
18 // Typecast pntr (void *) => (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(
24     msg_ptr, NUM_VER_SUPPORTED);
25 SET_MM_DEVICE_INFO_FIRST_SUP_PROTO_VERS(
26     msg_ptr, PROTO_VERSION_ONE);
27
28 // Send message to the DMTX task
29 ret_status = m_ici_io_ptr->send(&ici_msg_object);
30
31 // Check that message was sent successfully
32 if (ret_status != SUCCESS)
33 {
34  // Report problem when sending ICI message
35  sz_err_msg(MAJOR, SZ_ERR_MSG_ERR_OPCODE, __FILE__, __LINE__,
36             "DataMobilityRx processUnsupVer: failure sending "
37             " device info message to DMTX");
38 }
```

Figure 3. C++/ICI DM Communication

3.4.3 Automatic Memory Management

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 by guaranteeing safe storage management and reducing space leaks. 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.

3.4.4 Pragmatics

To aid rapid application development Erlang is supplied with the Open Telecom Platform (OTP) libraries [25]. The OTP include, *inter alia*, libraries, design principles, and productivity, profiling and debugging tools. A compiler [12] and a bytecode interpreter are both available open source for Erlang.

3.5 Very High-Level Distribution: GdH

Glasgow distributed Haskell (GdH) is a research language with very high-level distributed coordination. It was designed to investigate the construction of reliable distributed applications in high-level languages. Haskell [20] is the *de facto* standard non-strict functional language and the GdH implementation is based of the Glasgow Haskell Compiler, arguably the best Haskell implementation. GdH combines features of two other variants of Haskell, Glasgow parallel Haskell [26] and Concurrent Haskell, with some additional constructs [22].

GdH has the following features. It supports both parallel distributed computation using two classes of thread: stateless threads and stateful I/O threads. Processing Elements (PEs) are identified so a program can use resources unique to a PE, like a data source or a GUI interacting with a user. Both remote procedure call and remote evaluation distribution paradigms are supported. Remote procedure call is provided by the `revalIO` primitive:

```
revalIO :: IO a -> PEId -> IO a
```

and remote evaluation by the `rforkIO` primitive:

```
rforkIO :: IO () -> PEId -> IO ThreadId
```

As in Concurrent Haskell some communication and synchronisation is implicit: threads on one PE can share variables with threads on other PEs. Moreover, stateful threads can explicitly communicate and synchronise using *distributed* polymorphic semaphores (`MVars`). Higher-level constructs, like channels between threads on different PEs, 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.

GdH supports very high-level distributed coordination in several ways. Typically only a few key stateful objects are explicitly located, and the location of the large majority of stateless objects in a program is implicit. The communication and synchronisation associated with these objects is entirely implicit and managed by the sophisticated language implementation. Moreover, stateful computations are concisely expressed using abstractions like channels and higher-order monadic functions. For example a monadic map that distributes an action to a sequence of locations effectively performs a sequence of remote procedure calls:

```
mapM (\p -> (revalIO work p)) pes
```

4. Basis of Comparison

This section outlines the challenges posed by engineering of telecom software and describes the telecom components that are reengineered as the basis of our comparison between the distributed languages.

4.1 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, and application software. Rapid development and high levels of reliability and availability are key requirements. Telecoms providers aspire to 99.999% availability, which equates to little more than 5 minutes downtime a year, but this is rarely achieved.

The rapid production of robust telecoms software raises the following 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 typically consumes more than 50% of the software development effort. 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.

Currently many distributed telecoms systems are implemented in C with SDL on real-time operating systems with trends towards using C++/CORBA, JAVA/RMI. There is considerable interest in applying higher level techniques. One such technique is model driven software engineering, and UML 2.0 State Machines are a common model. Similarly, high level distributed programming languages are attractive because of the potential to reduce development time, and improve reliability and maintainability. Clearly the implementation technology must also meet the other functional requirements of telecom applications, e.g. real-time requirements.

4.2 Dispatch Call Controller (DCC)

The first component reengineered is a prototype dispatch call system developed at Motorola Labs in Illinois [15]. Dispatch call processing is a prevalent feature of many wireless communication systems. Managing the call processing with a distributed paradigm enables throughput to be scaled as system usage grows, with work dynamically distributed to the resources available. The requirements for the DCC model are derived from the technical report by Lillie [15] and from a set of functional requirements [24].

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. For the purpose of the study 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. More complete descriptions of the systems measured are available in [18].

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: one that sets up the call and generates simulated voice traffic, and another that simulates subscriber roaming by issuing hand-off messages to the fixed-end notifying it that the subscriber is now associated with a new base radio station. The fixed-end is simulated by a service dealing with both hand-offs and call requests.

The DCC software test platform comprises the subsystems described below and the overall architecture is shown in Figure 4. The hardware platform is a 32-node Beowulf cluster.

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

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

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

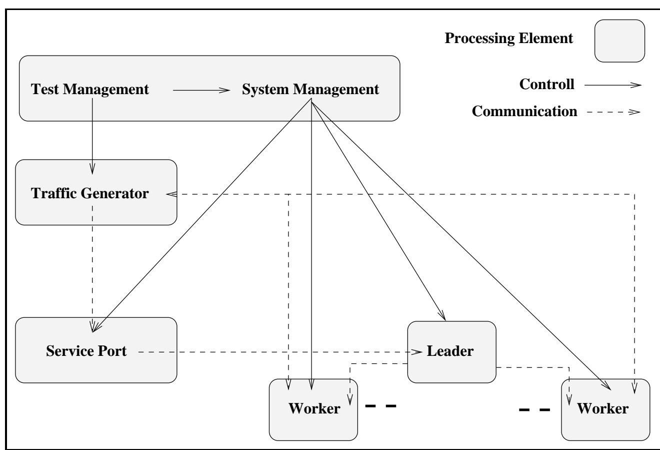

Figure 4. ERLANG DCC Architecture

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 elected, and re-elected after a failure, using a standard protocol.

4.3 Data Mobility Server (DM)

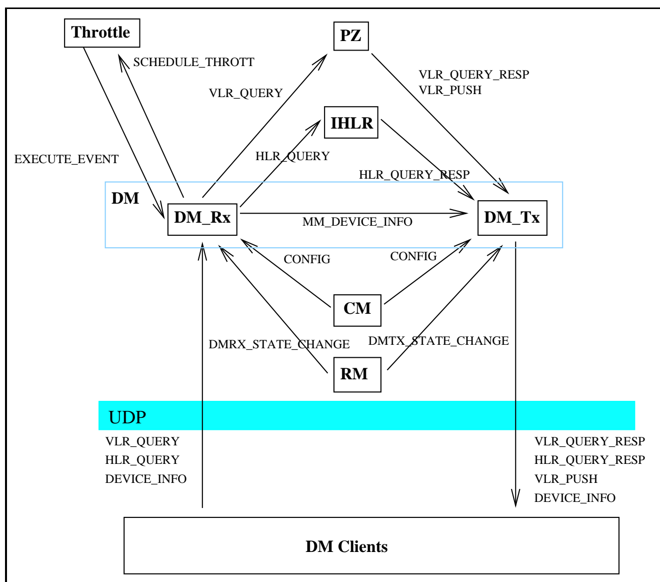

Figure 5. Abstract DM Architecture

The Data Mobility service is a small component of a radio communications subsystem (RCS) which is responsible for communication between the RCS and mobile data devices. The following DM description avoids using precise product names, and some details are made abstract to preserve commercial confidentiality. Both the RCS and DM were developed by Motorola as part of an existing product that follows an international standard.

Lang.	C++	IDL	**Total**	**Ratio**
C++/CORBA	13906	83	13989	42
ERLANG			2143	6
GdH			335	1

Table 1. DCC Code Sizes (SLOC)

The abstract DM architecture is shown in Figure 5, 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 using ICI. For brevity this is termed the C++/UDP implementation in the remainder of the paper.

The DM has not been implemented in GdH, but two ERLANG DM implementations have been constructed, one purely in ERLANG, and an ERLANG/C implementation that reuses some C DM libraries thus allowing the measurement of interoperation costs. Key aspects of the ERLANG DM architectures are as follows. There are four primary components: DME_Rx and DME_Tx are ERLANG receiver/transmitter processes and DME_Rx-drv and DME_Tx-drv are C receiver/transmitter drivers. The architecture combines Unix processes, C threads, and ERLANG processes. The ERLANG DMs interoperate with the same C RCS test harness as the C++ DM.

5. Application Size

This section investigates the impact of high-level distribution on application size by comparing the sizes of the DCC and DM components in ERLANG and GdH with the existing C++/CORBA and C++/UDP implementations. The significance of software size is well established: shorter programs are faster to produce [13, 23], and hence programmers working in higher level languages are more productive. The reduced development time crucially reduces time to market for the product. Moreover, shorter programs are easier to maintain, which is important as more than 50% of programming effort is expended on maintenance [13].

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 [16], and a good survey is available in [5]. 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 enabling cross-paradigm comparisons, in this case to compare functional and object-oriented programs.

5.1 Dispatch Call Controller

The sizes of the C++, ERLANG and GdH DCC implementations are reported in Table 1, together with the ratios between the sizes. The ERLANG implementation is a seventh of the size, and the GdH implementation is $1/42$nd of the size, of the C++/CORBA implementation.

While the language features contributing to the size differences are discussed in the following section, Table 2 exhibits additional factors contributing to the size differential. The table distinguishes between the *testing* component, the generic and reusable *platform* component, and the specific DCC *service* component of the implementations. While all three implementations meet the DCC functional requirements, the GdH implementation is less generic. That is, additional effort would be required to adapt it to meet similar functional requirements, and this is reflected in the relatively large service component. Secondly, while testing is a crucial part of application development, the sizes of the DCC testing components

Lang.	C++/CORBA			ERLANG			GdH		
	SLOC	%	No. Mod	SLOC	%	No. Mod	SLOC	%	No. Mod
Reusable Platform	13544	81%	30	1996	52%	22	305	67%	4
Specific Service	445	3%	5	147	4%	1	30	7%	1
Appln. Total	13989		35	2143		23	335		5
Testing Stats	868	6%	1	1687	44%	9	119	26%	1
Total	14857	100%	36	3830	100%	32	454	100%	6

Table 2. DCC Functional Analysis (SLOC)

Lang.	C/C++	ERLANG	**Total**	**Ratio**
C++	3101		**3101**	7.8
ERLANG/C	247	616	**863**	2.2
ERLANG		398	**398**	1

Table 3. DM Code Sizes (SLOC)

should not be compared as the three DCC implementations have not been tested to the same level. That is, the C++/CORBA and GdH implementations have been lightly tested, whereas the ERLANG implementation has been substantially tested.

5.2 Data Mobility Server

The size of the C++ and ERLANG DM are reported in Table 1, and the sizes of the DMs are depicted in Figure 6. As for the DCC, the pure ERLANG DM implementation is significantly smaller than the C++/UDP implementation. Even the interoperating ERLANG/C implementation is 3.6 times smaller than the C++/UDP implementation.

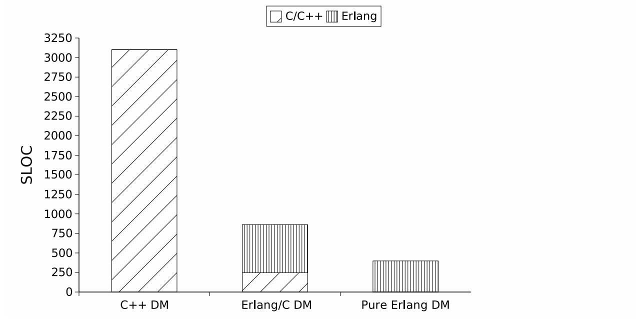

Figure 6. Source Code Sizes

5.3 Size Discussion

The GdH DCC is far smaller than both the ERLANG and C++/CORBA DCCs reflecting the very high level specifications of both computation and distributed coordination. The ERLANG DCC and DM are less than 1/6th of the size of the corresponding C++ implementations reflecting its high-level coordination. The ERLANG result is consistent with other measurements [28], and with developer folklore in companies like Ericsson, T-Mobile and Nortel. We conclude that high-level distributed language constructs reduce the application size of realistic components. Moreover the GdH DCC results suggest that very high-level distribution constructs reduce application size still further. The following section investigates the language constructs that contribute to the size reduction.

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 4. C++ DM Code Proportions

Code Type	ERLANG Total	ERLANG/C Total	ERL./C ERLANG Part	ERL./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 Mgmnt	0.0%	3.2%	0.0%	11.3%
Type Decls	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 Mgmnt	9.5%	5.6%	5.5%	5.7%

Table 5. ERLANG DM Code Proportions

6. Language Feature Impact

This section attributes size reductions to specific language features by analysing the DM component code. With a total of 19K lines of code, the DCC is too large to analyse conveniently. Figure 7 and Tables 5 and 4 compare the C++ and ERLANG DM code. For simplicity the following discussion compares the C++ DM only with the pure ERLANG DM. When interpreting the percentages in these results, the reader should recall that the ERLANG DM is 1/7th of the size of the C++ DM.

The results show that the primary language features contributing to the reduction in code size are as follows.

- ERLANG's **sophisticated fault tolerance** mechanisms mean that the programmer can code for the successful case. Hence, there is far less defensive code in the ERLANG implementation: 0.5% as opposed to 25.3%
- ERLANG's **high-level communication** greatly reduces the effort of specifying communication, e.g. buffering, marshalling and error-checking: 15.1% as opposed to 23.9%.
- ERLANG's **garbage collection** greatly reduces the memory management coding effort: 11.6% as opposed to 0%.

Of crucial importance to product development teams, 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.

7. Coordination Code Size

This section investigates the hypothesis that a language with high-level coordination reduces the amount of coordination that the programmer must specify. While the hypothesis is widely believed, as far as we know this is the first evidence for a realistic, i.e. 15K line software component.

Table 6 reports the amount of coordination code in the ERLANG, GdH and C++/CORBA DCC implementations. The measurements exclude declarations for coordination. While the percentage coordination for C++/CORBA and ERLANG is similar, the amount of

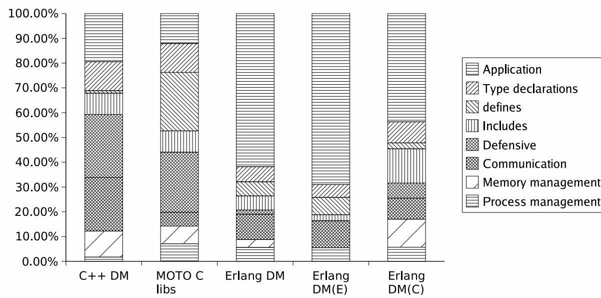

Figure 7. Source Code Breakdown

Part	SLOC	Percentage	Modules
C++/CORBA	2812	19%	13
ERLANG	881	23%	9
GdH	44	10%	4

Table 6. DCC Coordination Code Sizes

	ERLANG DCC			ERLANG FT DCC			Size Increase		
Part	SLOC	%age	Mod.	SLOC	%age	Mod.	SLOC	%age	Mod.
Total	3830	100%	32	4882	100%	38	27%		19%
Platform	1996	52%	22	2994	61%	26	50%	9%	18%
Service	147	4%	1	147	3%	1	0%	-1%	0%
Testing	1687	44%	9	1741	36%	11	3%	-8%	22%
Coord.	881	23%	9	1933	40%	28	119%	17%	210%

Table 7. Original and Fault Tolerant ERLANG DCC Code Analysis

code is significantly less, i.e. 881 SLOC as opposed to 2812. The very high-level coordination constructs in GdH reduce the coordination specification cost still further to just 10% or 44 SLOC. We conclude that, as expected, high-level coordination constructs reduce the amount of coordination that the programmer must specify, and that very high-level constructs reduce it yet further.

8. Cost of Adding Fault Tolerance

This section investigates the cost of adding robustness to a realistic distributed system using a language with sophisticated fault tolerance. Our investigation uses the ERLANG fault tolerance mechanisms outlined in section 3.4.

The C++/UDP DM includes 25.3% defensive code (Table 4), and our Motorola collaborators assure us that this is relatively low for telecoms components. In contrast the ERLANG DM includes just 0.5% defensive code (Table 5). Moreover, we have elsewhere demonstrated that the ERLANG DM is *resilient*: sustaining throughput at extreme loads and automatically recovering when load drops. In contrast the C++/UDP DM fails catastrophically when overloaded [17, 18]. Of course the C++ DM could be reengineered to be resilient, typically by declining excess requests, and also to recover. However this would make the program even larger, more complex, and harder to validate.

To quantify the cost of adding fault tolerance we have constructed a second, fault tolerant DCC, the ERLANGFT DCC. We have elsewhere demonstrated that the ERLANGFT DCC exhibits high levels of *availability*: remaining available despite repeated and multiple hardware and software failures, and *dynamic reconfigurability*: with throughput scaling near-linearly when resources are added or removed [17]. As the C++/CORBA DCC is not fault tolerant it is compared for size with the original ERLANG DCC version in Table 1.

Part	Lines of Code	Percentage
ERLANG FT	164	3%
C++	3574	24%
GdH	137	30%

Table 8. Type Declarations

Part	Lines of Code	Percentage
total	4882	100%
dynamic tests	687	14%

Table 9. ERLANG FT DCC Dynamic Type Tests

The sizes and functions of the ERLANGFT and original ERLANG DCCs are reported in Table 7, where the last line compares the sizes of the coordination functionality in each implementation. The 8th column of the table show that there is a modest 27% increase in total application size, and that most of the increase is in the reusable platform (50%). The last row of the table shows that there is a dramatic increase in the amount of coordination code (119%), moreover the coordination code has become pervasive, increasing by 210% from just 9 out of 32 modules in the original ERLANG DCC to 28 out of 38 modules in the ERLANGFT DCC.

We conclude that distributed languages with sophisticated fault tolerance, like ERLANG, greatly facilitate the construction of robust, i.e. resilient, highly-available and dynamically-reconfigurable, systems. Moreover robustness is introduced for a modest cost: an increase in total application size of 0.5% for the DM and 27% for the DCC. The additional fault tolerance code primarily specifies coordination, and coordination becomes pervasive in the application.

9. Type System Impacts

Distributed software places demands on the type system: components of a distributed system must typically be separately typed, often statically, while messages between components must be dynamically typed. The languages in our study have very different type systems: ERLANG is dynamically typed, C++ is weakly typed with parametric and subtyping polymorphism, and GdH is strongly statically typed with Hindley-Milner and parametric polymorphism. This section investigates some of the impacts of the different type systems on realistic distributed software.

Table 8 reports the number of type declarations in each of the DCC implementations. The dynamic typing in ERLANG means that it contains very few declarations, just 3%. In contrast the C++ DCC contains the greatest number of type declarations, 3574, and a far higher percentage than in ERLANG. This represents both the static typing, and limited use of type inference in C++. Finally, the table shows that while the GdH DCC contains relatively few type declarations, just 137, these represent the highest percentage of the code, 30%. Many of the GdH type declarations could be inferred and hence are not necessary, but are commonly included as a programming discipline. It is commonly believed that the additional safety, static checking, and self documentation provided by strong static typing more than compensate for this substantial overhead.

Table 9 reports the number of dynamic type checks in the ERLANG FT DCC. It reveals a significant proportion of dynamic type tests, with one source line in 7 making a type test. We infer that dynamic typing is likely to incur a significant performance overhead for this component, and suggest that this is typical of other distributed components.

10. Conclusions

10.1 Summary

We have investigated the impact of high level distributed programming language constructs on the engineering of two telecoms software components. Our investigation compares GdH, a language with very high-level distribution, and ERLANG, a language with high-level distribution, with conventional distributed technologies: mid-level C++/CORBA and low-level C++/UDP. Let us return to the research questions from the introduction.

Q1 Do high-level distributed language constructs reduce application size? Reflecting it's high-level distribution, the ERLANG DCC and DM are less than 1/6th of the size of the C++ DM. The ERLANG result is consistent with other measurements [28], and with developer folklore. We conclude that high-level distribution language constructs significantly reduce the application size of realistic components. With very high level coordination, the GdH DCC is far smaller than both the ERLANG and C++/CORBA DCCs and this suggests that very high-level distribution constructs reduce application still further (Section 5).

Q2 What high-level distributed language constructs impact application size? Our investigation identified three primary high-level language constructs that reduce distributed programming effort: sophisticated fault tolerance case saves 27%, high-level communications save 22%, and automatic memory management saves a further 11% (Section 6).

Q3 What is the impact of high-level coordination? While the percentage of coordination code in the ERLANG and C++/CORBA DCCs is similar, approximately 20%, the amount of code is significantly less: 881 SLOC as opposed to 2812. The very high-level coordination constructs in GdH reduce the coordination specification cost dramatically further to just 10%. We conclude that, as expected, high-level coordination constructs reduce the amount of coordination that the programmer must specify, and that very high-level constructs reduce it yet further (Section 7).

Q4 What is the impact of sophisticated fault tolerance constructs, and what are the costs of introducing robustness? Robustness is introduced for a modest cost using advanced fault tolerance. The C++/UDP DM includes 25.3% defensive code and yet fails to provide key robustness capabilities, e.g. resilience to overload. In contrast, the ERLANG DM includes just 0.5% defensive code and provides additional robustness, including resilience. Adding fault tolerance increases the size of the DCC by just 27%, and much of the additional effort is expended in the reusable platform (50%). Significantly, almost all of the additional code is coordination, i.e. an additional 119%, and coordination becomes pervasive throughout the application, appearing in 28 out of 38 modules (Section 8).

Q5 What are some of the impacts of the type systems on distributed software engineering? The DCC in dynamically-typed ERLANG requires relatively few type declarations, just 3%, but induces substantial amounts of dynamic type checks, e.g. 1 source line in 7. The DCC in statically-typed C++ requires the greatest number of type declarations, 3574 and 24%, reflecting both static typing and limited type inference. The DCC in GdH, reflecting the common idiom of Hindley-Milner polymorphism, contains the the greatest percentage of type declarations, 30%, yet the least number of type declarations (Section 9).

10.2 Discussion

It can be argued from the results in Sections 5- 7 that a language with very high-level distribution, like GdH, gives the greatest benefits for distributed software development. These results should, however, be viewed with some caution. GdH is a research language and lacks a production-quality implementation. More significantly, GdH's distributed paradigm is limited in a number of ways. GdH uses a distributed virtual shared-memory model, so distributed performance will not scale as well as a distributed-memory model like ERLANG. GdH has conventional fault tolerance using distributed exceptions, where other languages including ERLANG have more advanced models. GdH supports only closed systems: i.e. while an arbitrary number of processes can be created on an arbitrary number of processors, all of the distributed processes are part of a single program and no new programs nor new processors can be added (or removed). Finally, GdH is lazy and hence it's harder to statically predict program performance, often a crucial aspect of distributed systems.

We conclude that high-level distributed language constructs can aid the rapid production of realistic robust systems. Moreover, we have presented some evidence that very high-level distributed language constructs further aid the rapid production of realistic systems. The high-level constructs dramatically reduce application size, thereby reducing development time and aiding maintenance. Sophisticated fault tolerance, high-level communication and automated memory management are major contributions to reducing the development effort. High level constructs greatly reduce the amount of coordination the programmer must specify, and very high-level constructs reduce it dramatically. Robustness is introduced for a modest cost using sophisticated fault tolerance.

In ongoing work we are working with a product group within Motorola to reengineer a substantial distributed system in ERLANG. We are also investigating a generic toolkit for constructing ERLANG wrappers to make existing components written in conventional languages robust and scalable.

Acknowledgments

We gratefully acknowledge research funding from the UK EPSRC (GR/R88137) and from Motorola UK Research Labs.

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

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

[6] 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.

[7] S. Gorlatch. Send-receive considered harmful: Myths and realities of message passing. *ACM Transactions on Programming Languages and Systems*, 26(1):47–56, 2004.

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

[9] 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.

[10] 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.

[11] 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, Dec. 2003.

[12] E. Johansson, M. Pettersson, K. Sagonas, and T. Lindgren. The development of the HiPE system: Design and experience report. *Software Tools for Technology Transfer*, 4(4):421–436, August 2003.

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

[14] N. Kamenoff and N. Weiderman. Hartstone distributed benchmark: Requirements and definitions. In *Proceedings of the 12th Real Time Systems Symposium*, pages 199–208, San Antonio, Texas, USA, 1999. IEEE.

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

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

[17] J. Nyström, P. Trinder, and D. King. High-level Distribution for the Rapid Production of Robust Telecoms Software: Comparing C++ and ERLANG. *Concurrency and Computation: Practice and Experience*. To Appear.

[18] J. Nyström, P. Trinder, and D. 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.

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

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

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

[22] R. Pointon, P. Trinder, and H.-W. Loidl. The Design and Implementation of Glasgow distributed Haskell. In *Proceedings of the International Workshop on Implementing Functional Languages (IFL'00)*, LNCS 2011, pages 101–116, Aachen, Germany, Sept. 2000.

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

[24] L. Rittle. Distributed Dispatch Architecture Project (Functional Requirements). Technical report, Land Mobile Products Sector Research, Shaumburg, Illinois, 1998.

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

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

[27] U. Wiger. Industrial-Strength Functional Programming: Experiences with the Ericsson AXD301 Project. In *Proceedings of the International Workshop on Implementing Functional Languages (IFL'00)*, Aachen, Germany, Sept. 2000. Presentation Only.

[28] 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.