*J. Functional Programming* **10** (1): 91–120, January 2000. Printed in the United Kingdom 


# *Distributed communication in ML*

MARK HAYDEN*

*Compaq Corp., Systems Research Center,*
*130 Lytton Avenue, Palo Alto, CA 94301, USA*

**Abstract**

We present our experience in implementing a group communication toolkit in Objective Caml, a dialect of the ML family of programming languages. We compare the toolkit both quantitatively and qualitatively to a predecessor toolkit which was implemented in C. Our experience shows that using the high-level abstraction features of ML gives substantial advantages. Some of these features, such as automatic memory management and message marshalling, allowed us to concentrate on those pieces of the implementation which required careful attention in order to achieve good performance. We conclude with a set of suggested changes to ML implementations.

**Capsule Review**

This paper reports on the author's experience in implementing Ensemble, a network group communication system. The experience is especially interesting because (a) the communication system is 'serious' in the sense that it supports several real-world applications and is an improvement of a previous commercial-grade system, (b) it is written almost entirely in Ocaml, leading to dramatic reductions in code complexity (and improvements in the programmer's quality of life!) and (c) despite the fact that it was written in Ocaml, Ensemble performs significantly better than the earlier C-based version. The paper describes what was good and what was bad about using Ocaml for systems-level programming, and in the case of the bad, the methods by which the problems were overcome. Certainly, many ML (and typed functional programming) researchers will find the experience to be informative and inspirational; and perhaps a few systems researchers might become a bit more open-minded about ML.

## 1 Introduction

This paper presents our experiences building a group communication system in a dialect of the ML programming language. We believe our experiences are interesting for several reasons. First, the system is the third generation of its kind built by our research group. The previous systems were developed in the C programming language; we draw comparisons between three similar efforts, the last two of which primarily differ in the use of ML rather than C. Second, all of these systems were

* This work was completed while the author was at the Cornell University Computer Science Department. The work was supported by DARPA/ONR contract N0014-96-1-10014 and ARPA/RADC contract F30602-96-1-0317.

multi-year development efforts and are big enough to be able to test their languages' (in particular, ML's) ability to support systems 'in the large'. Third, we also address questions regarding the usability of ML for systems style programs. Finally, the change to a new language was motivated by demonstrable advantages, including performance, of an implementation in ML over that of one in C.

The impact of ML on our work forms the thread of this paper. The questions addressed include: how much (and in what ways) did the use of ML help or hinder? Also, what could be added to ML implementations to better support systems work? For instance, do they need support for direct access to low-level data types such as untagged word arrays or data structures defined in C? These questions are briefly addressed below and then in more detail in the remainder of the paper.

### *1.1 Background*

Since 1985, the distributed systems research group at Cornell has developed three generations of group communication toolkits. The first, ISIS (Birman and van Renesse, 1994), was a pioneering group communication system that eventually grew into a successful commercial product. The second generation, Horus (van Renesse *et al*., 1996), was a reimplementation of ISIS based on a more flexible architecture with dynamically configurable protocol layers. Both ISIS and Horus were written in C. Ensemble, the third generation, is a reimplementation of Horus in ML. All three systems are similar in that they support the same basic process group communication model, but all have important differences. Architecturally, Ensemble and Horus are closer than Horus and ISIS. The similarity of Ensemble and Horus is important because it allows us to draw comparisons between them in order to understand the impact of the respective programming languages. However, care must be taken in making comparisons because of the large number of factors involved. This paper does not present details of the architectures except where necessary in order to demonstrate the impacts of the relative programming languages.

All three of these systems have been used in a variety of applications. As a commercial product, ISIS appears in many real-world systems in current use, including several stock exchanges, a major air traffic control system, VLSI fabrication process planning software, and other significant, critical applications. Horus was introduced as a more flexible, better structured alternative to ISIS, and might be commercialized by Stratus (the owner of ISIS). The most recent system, Ensemble, has a number of applications being used in the Cornell Computer Science department, including a virtual private network service, a partitionable process management system, and a distributed audio server. Also, other research groups are actively developing applications using Ensemble's C++ interfaces, and some ISIS users are now switching over to use Ensemble. Given time, we believe that Ensemble will penetrate the same types of critical environments as did ISIS.

Horus and Ensemble are similar because Ensemble was initially intended as a *reference* implementation of Horus for use in prototyping changes to Horus. Their architectures began diverging only after our research group elected to switch over to Ensemble entirely. As Ensemble matured, we began to question the need to maintain

two versions of the system, and eventually Ensemble became the production as well as reference implementation. The primary cause for this change of thinking was the realization that a reimplementation in ML could achieve the same or better performance as Horus in C. This was a surprise to us. The structural changes we made in Ensemble enabled a sequence of optimizations not possible in the original Horus system. These optimizations make Ensemble the lowest latency reliable group communication system currently available. The use of ML facilitated the discovery and implementation of these optimizations, which are described by Hayden and van Renesse (1997) and formalized by Kreitz (1997) and Hayden (1998).

The other major factors that led us to adopt Ensemble were related to its cleaner architecture and increased flexibility. We find that Ensemble better exposes real structure in the system and that this has repeatedly revealed new approaches for design improvements, while also making it easier to experiment with these new approaches. An example of the improvements aided by Ensemble's architecture is the decomposition of protocol layers in Ensemble and Horus. Both systems divide high-level protocols into small, composable protocol layers. In Ensemble, protocol stacks tend to have nearly twice as many protocols as Horus, with comparable protocol layers being around an order of magnitude smaller. The decrease in the size of the protocol layers is very important because the protocols are the most complex part of the system. Having more layers each of which are much smaller has meant that there is an increase in modularity and code reuse, and an overall decrease in code size over the version in C. Both the improved decomposition of the layers and the decrease in size are important when experimenting with new design approaches because they provide increased modularity and flexibility.

### *1.2 Impact of ML*

Though our focus is on our work using ML, we believe many of our experiences would have been similar had we chosen to use another programming language with similar features such as strong static type checking, automatic marshalling, automatic memory management, higher-order functions, polymorphism, and exceptions. One of the major reasons for initially choosing ML was that we wished to investigate the use of theorem provers for verifying layered protocol architectures. A subset of ML is straightforward to import into NuPRL (Constable *et al.* , 1986), a theorem prover developed at Cornell that we are using for verification, and this led us to use ML (Kreitz, 1997). In this paper, we do not discuss our work with NuPRL, but focus on the other effects of ML.

ML constructs and common implementation mechanisms such as exceptions, garbage collection, and higher-order functions are useful but typically have a cost to performance in the language implementation. Ensemble extensively uses these features, but in a restricted fashion. To avoid performance costs, we carefully coded Ensemble so that the features are only used in ways that are easy to optimize or do not affect performance.

In the case of memory management, automatic garbage collection was initially used for all data structures. This greatly improved program development time and

correctness. However, it later became necessary to introduce explicit management for payloads of messages because the garbage collector caused memory fragmentation and high processing overhead. Having implemented several similar previous systems, we can point out that the need to specially manage message buffers is not particular to Ensemble. In both Horus and ISIS, careful treatment of message buffers was crucial to performance, and both systems developed complex, highly optimized subsystems for managing them. The difference between the three systems is that all data structures are explicitly managed in Horus and ISIS, whereas in Ensemble only buffers are.

The use of formal module systems has been suggested as an important reason for using ML in layered communication systems (Biagioni, 1994). Ensemble uses modules for separate compilation and this speeds development. However, we do not make use of module composition (functors) for protocol layer composition because Ensemble composes protocols at run time. Functors are second class objects in ML and must be composed at compile time, which is incompatible with the needs of Ensemble.

### *1.3 Related work*

Related work has been done in the Fox project (Biagioni *et al*., 1994) which demonstrated the use of ML for systems programming. They developed a complete TCP protocol stack in ML[1] that interfaces very closely with the network. However, Fox and Ensemble differ significantly. First, the Fox project implements TCP, which is a standard protocol, and so is constrained in many ways that Ensemble is not. For instance, TCP has fixed header formats that Fox TCP must adhere to, whereas Ensemble is free to set its own header formats and to change them as the system evolves (this issue is discussed at length in section 5.1). In addition, the Fox design is deliberately very similar to TCP implementations in C because the developers wished to show that systems can be built in ML in a fashion similar to C. Ensemble, on the other hand, was not restricted in this fashion. One result of these differences was that Ensemble was able to achieve better performance than the implementation of Horus written in C, whereas the Fox TCP implementation is slower than implementations of TCP in C.

Other related work has been done with Erlang (Armstrong *et al*., 1996; Hausman, 1994; Marlow and Wadler, 1997). Erlang, a product of Ericsson, is a functional language designed to support distributed telephone switching software. A number of impressive telecommunications products have been built using Erlang and they have found many of same advantages of using functional languages for distributed communication that we have. Erlang does not support static type checking, although there are several efforts underway to add this to the language. The approach with Erlang was to design a new language with support for distribution, whereas our approach has been to build libraries in an existing language.

[1] Fox uses Standard ML of NJ, whereas the implementation of ML we use is Objective Caml.

### 1.4 Contents

The rest of this paper proceeds as follows. Section 2 describes Objective Caml, the implementation of ML we use. Section 3 gives an overview of group communication and the architectures of Horus and Ensemble. Section 4 compares the implementations of Horus and Ensemble. Section 5 describes how messages are represented in Ensemble. Section 6 addresses issues with message buffers, which we needed to manage explicitly. Section 7 shows how inlining allows systems style applications to make extensive use of abstraction barriers with very little cost. We then conclude with a summary of our lessons and a list of features that we feel are missing from implementations of ML.

## 2 Objective Caml

We use the Objective Caml (Ocaml) system (Leroy, 1997) which implements its own sub-dialect of the CAML (Weis andLeroy, 1993) dialect of ML. Although this paper attempts to be general, it is important to distinguish between the ML family of programming languages and the particular implementation that we use. For instance, Standard ML is a language, Standard ML of NJ and Harlequin ML Works are implementations, and Ocaml is both a language and an implementation. The languages and implementations differ widely in number of ways. In this section, we briefly describe the Ocaml system, some features particular to Ocaml, and our experiences with it, both positive and negative.

### 2.1 Portability

The Ocaml system is actually two compilers. The first is a bytecode code compiler that provides rapid compilation, platform-independent bytecode, and good performance. The second is a native code compiler that gives slower compilation but generates higher-performance code. The compilers are interchangeable and run on a large number of platforms, including Windows NT, Windows 95, and most variants of UNIX. Porting Ensemble to new platforms has usually not involved modifying ML code, but revolved around issues outside of the control of the compiler (such as incompatibilities in the 'make' program). The native code compiler provides very good performance. It gives efficient support for curried functions and support for inlining within and across module boundaries. We give a detailed example of the optimizations in section 7. Other notable features of the system are a large library of UNIX system calls, support for automated marshalling of data structures, and features for object-oriented programming (although Ensemble does not use Ocaml's object-oriented features).

### 2.2 Performance considerations

In designing Ensemble, we were careful to restrict the use of certain features that can hurt performance. As an example, consider higher-order functions. They have the

https://doi.org/10.1017/S0956796899003275 Published online by Cambridge University Press

problem that their use often requires allocation of closures (closures are dynamically generated function objects). Higher-order functions are used extensively in Ensemble, but only so that closures are not allocated in the normal case. Two techniques were used to achieve this. The first was a phased approach where closures are created when protocol stacks are initialized, but not during their normal execution. This technique is similar to one described by Biagioni *et al.* (1994). The second way was to use higher-order iterators for data structures such as list and array iterators. These are efficient because inlining of the iterator can eliminate closure allocation. However, the Ocaml compiler currently does not do this so we currently hand-inline such iterators for the 20 or so occurrences in the common Ensemble execution paths, so closures are only created outside common execution paths and do not affect the performance of Ensemble.

### *2.3 Memory management*

Ocaml supports garbage-collected memory management. Although some ML implementations are perceived to require large amounts of memory (MacQueen, 1993), the Ocaml system is known for its efficient use of memory and this has not been a problem for us. Ocaml uses a generational garbage collector with a stop-and-copy minor heap and an incremental mark-and-sweep major heap. Our experiences with the garbage collector have been positive, with two exceptions. The first is that there is no support for compacting the major heap, which means that long-running programs never release memory from the major heap. This causes problems with highly available server applications that run for weeks at a time (and longer)[2]. The other exception is that the major heap does not do a good job of managing large objects and tends to fragment over time. This problem was observed with the message buffers that Ensemble uses and eventually caused us to manage them through explicit reference counting (see section 5.2 for more details). In summary, we found the Ocaml garbage collector extremely useful for almost all our data structures, but for some we had to take over and manage them ourselves.

### *2.4 Interoperability*

Ocaml provides support for easily interfacing with C programs. ML programs can issue cross-language calls to C functions and vice-versa. In both cases, exceptions are handled correctly across any number of calls into and out of C. ML objects can incorporate pointers to C objects outside the heap and C code can declare references to ML objects in the heap. With this support, we implemented a C interface to the Ensemble system. This required writing a set of C stub routines for calling into Ocaml code. In addition, at our prodding, the Ocaml implementors added the capability to generate C libraries from ML programs. This allows Ensemble to be built as a normal C library and linked with C programs. The Ensemble library can

[2] The next version of Ocaml, which has been released since writing this, now supports compaction. This was also added in part because of the problems we reported.

then be treated as a black box from C code which does not need to know that ML was used.

Although it is not difficult to link ML and C programs in this manner, we have noticed that memory errors in C programs can easily corrupt the ML heap (for instance by modifying dangling pointers that happen to point into the ML heap), which then usually causes an entire process to crash. C programs by chance may not access their own corrupted data structures, but the ML heap is regularly traversed by the garbage collector, so corruption of the heap is likely to result in a process failure. From the point of view of a C developer, even though the bugs in such cases are in the C code, the use of ML makes the entire program more 'fragile' and makes tracking down problems in the C portion more difficult. This problem is compounded by the fact that many C debugging tools (such as Purify) are unable to handle the ML heap. In the case of Ensemble, we avoid this problem by providing two versions of the C interface that appear identical to the application. The first, the 'inboard' version, includes Ensemble and the ML run-time system in the C process. This version provides the best performance, but exhibits fragile behavior when the C program is buggy. With the second, the 'outboard' version, the C program and Ensemble execute in separate processes which communicate via UNIX pipes. This version is used while debugging applications because it isolates the ML heap from application errors.

### *2.5 Debugging and profiling*

Until recently, Ocaml did not provide a debugger, which occasionally made debugging difficult. However, we found many of the hardest problems to debug in C, such as memory errors, are prevented by the ML type checker, and so the impact of a missing debugger was somewhat reduced. A continuing problem, however, is the difficulty of profiling memory behavior of Ocaml programs (normal C tools can be used for standard execution profiling). While predicting the operations that cause memory allocation is usually easy in Ocaml programs, it is much more difficult to get a good picture of the overall memory allocation patterns in programs. Other systems, such as Harlequin's Standard ML environment (Harlequin, 1996), provide support for profiling memory usage, so this kind of support is certainly possible.

### *2.6 Summary*

Our experience with Ocaml has been generally positive. It has provided a stable platform for us, and when there have been bugs in the system, the implementors have been quick to respond with fixes. Our success with ML is due in no small part to the excellent job of the Ocaml developers.

## 3 Group communication and layering protocols

In order to compare Horus and Ensemble, it is useful to give brief overviews of group communication and their architectures. Group communication is a generalization

https://doi.org/10.1017/S0956796899003275 Published online by Cambridge University Press

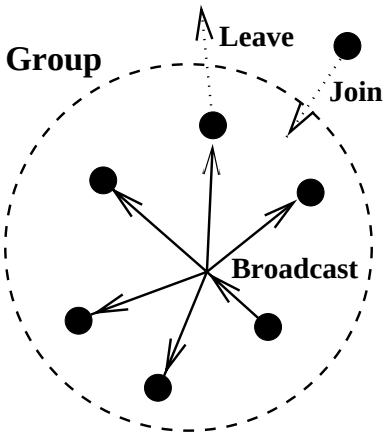


Fig. 1. Groups support operations for endpoints to join and leave. In addition, processes broadcast messages to a group.

of point-to-point protocols (such as TCP/IP) from communication among a pair of *endpoints* to communication among groups of them. Communication is always done in the context of groups, and there are operations for endpoints to join groups, to leave them, to send point-to-point messages to other members, and to broadcast messages to the entire group (see Figure 1).

Protocols such as TCP are useful largely because of the properties provided to applications. In the case of TCP, these properties include point-to-point, reliable, FIFO delivery of messages, flow control, and failure detection. With secure sockets (Atkinson, 1995), applications also get security properties. Similarly, group communication protocols provide a variety of properties. In fact, the properties that can be provided for groups are much richer than for pairs of processes because group-based applications have more structure. For instance, there are ordering properties on broadcasts (FIFO, Causal, Total), state transfer, network partitioning detection and healing, and client-server management. See Table 1 for a list of many of the properties supported by Ensemble. In addition, there are many different protocols that satisfy these properties (Birman, 1996).

One of the goals of Ensemble is to provide flexible support for a large collection of properties. To achieve this goal, Ensemble uses a layered architecture, in which small protocol layers are composed to create protocol stacks (Hayden and van Renesse, 1997). This architecture is similar to the one developed for Horus. An application requests a set of high-level properties and Ensemble constructs an appropriate stack of protocol layers from its growing library of more than 50 protocols. See figure 2 for the protocol stack generated from the default Ensemble properties. Unlike Horus, Ensemble determines the order in which to stack layers. Also, an application can change the desired properties on the fly and Ensemble will automatically switch to a protocol stack that meets the new demands.

Group communication is often used to build fault-tolerant applications through a set of properties called *virtual synchrony* (Birman and Joseph, 1987). Virtual

Table 1. *Some of the properties currently supported by Ensemble*[a]

| *Property* | *Description* |
|---|---|
| Agree | agreed (safe) delivery |
| Auth | authentication |
| Causal | causally ordered broadcast |
| Cltsvr | client-server management |
| Debug | adds debugging layers |
| Evs | extended virtual synchrony |
| Flow | flow control |
| Frag | fragmentation-reassembly |
| Gmp | group membership properties |
| Heal | partition healing |
| Migrate | process migration |
| Privacy | encryption of application data |
| Rekey | support for rekeying the group |
| Scale | scalability |
| Suspect | failure detection |
| Switch | protocol switching |
| Sync | group view synchronization |
| Total | totally ordered broadcast |
| Xfer | state transfer |

[a]These draw from a large background of research on group communication.

| | |
|---|---|
| Top | top-most protocol layer |
| Heal | partition healing |
| Switch | protocol arbitration and switching |
| Migrate | process migration |
| Leave | reliable leave |
| Inter | multi-partition view change |
| Intra | single partition view change |
| Elect | leader election |
| Merge | reliable merge protocol |
| Slander | failure suspicion sharing |
| Sync | view change synchronization |
| Suspect | failure detector |
| Stable | broadcast stability detection |
| Appl | application representative |
| Frag | fragmentation-reassembly |
| Pt2ptw | point-to-point window flow control |
| Mflow | multicast flow control |
| Pt2pt | reliable, FIFO point-to-point |
| Mnak | multicast NAK protocol |
| Bottom | bottom-most protocol |

Fig. 2. Sample protocol stack. This is the protocol stack created when using the default Ensemble properties {Gmp, Sync, Heal, Migrate, Switch, Frag, Suspect, Flow}. Ensemble provides a facility for translating from abstract properties to concrete protocol stacks.

https://doi.org/10.1017/S0956796899003275 Published online by Cambridge University Press

synchrony guarantees that all members in a group see the same progression of membership changes or *views*. When failures are suspected (usually through timeouts), the remaining members of the group mark the members as faulty and install a new view that does not contain the suspected failed members. Virtual synchrony also guarantees that all messages are delivered in the same view in which they were sent, and all endpoints that see the same two consecutive views deliver the same set of broadcasts between the views. When used with a totally ordered broadcast protocol (Ensemble has several such protocols in its library), virtual synchrony can be used to build fault-tolerant objects through the state-machine approach to replication (Birman, 1996; Schneider, 1990). Consider a replicated database in which the processes of a group form a set of distributed servers. The state is replicated across all the members in the group, and deterministic updates are disseminated to the group through total-ordered broadcasts. If all replicas begin in the same state and receive the same updates in the same order, then they all progress through the same set of states. Such an approach exploits many of the properties of Ensemble and substantially simplifies the development of distributed, fault tolerant applications.

## 4 Comparing C and ML implementations

The similarities of Horus and Ensemble allows us to draw a variety of comparisons between the systems in order to better understand the impact of ML. We begin by discussing why it is reasonable to compare the two systems, and then present a variety of comparisons, from more to less concrete. Throughout these comparisons, keep in mind that for all their similarities, Ensemble and Horus are different in many ways. Ensemble supports almost all the functionality that Horus does and many things Horus does not, but the functionality of Ensemble is still neither a superset nor a subset of Horus. The design of Ensemble embodies many lessons we learned from Horus. Were we to rewrite Ensemble in C, we would probably arrive at a system closer to Ensemble than Horus, even with differences in the programming languages. Also note that both Horus and Ensemble are highly modular systems and each have many different configurations. In comparing them, we have attempted to match comparable configurations where possible.

### *4.1 Development times*

Both Ensemble and Horus were developed primarily by single (though different) programmers and contributions from the research group were primarily made in the form of additional protocol layers or interfaces to support additional programming languages. Horus was actively developed for two years. At the time of writing, Ensemble has been under development for 1.5 years.

### *4.2 Language interfaces*

Both Ensemble and Horus are intended to be able to fit a variety of interfaces and to be used from many programming languages, so a consideration in our switching

to use ML was the question of how accessible the system would be to programs written in C. As described above in section 2, Ocaml provides adequate support for interfacing with C. In addition to C, C++, Tcl/Tk, and CORBA (Maffeis, 1995), which are supported by Horus, Ensemble also supports Smalltalk, Ada, and (of course) ML.

### *4.3 Supported platforms*

Both Ensemble and Horus seek to be largely platform-independent. Horus is supported on a variety of UNIX platforms and a smattering of other operating systems such as Chorus and Mach. Supporting new platforms for Horus requires writing some low-level system calls to access platform-specific thread and messaging operations. Ensemble runs on all platforms supported by Ocaml, including practically all UNIX platforms and Windows 95 and Windows NT. The use of ML has meant for us that porting issues are largely left to the ML compiler. For instance, there are some platforms (IBM AIX and Hewlett Packard HPUX) that were too bothersome to support for Horus that are supported by Ensemble because it is no trouble to do so.

The Ensemble software distribution includes a pre-compiled ML bytecode library that can be used on all supported platforms. A user downloading Ensemble on any platform merely compiles the demonstration programs (or their own programs) and links with the platform-independent bytecode libraries we provide. The platform-dependent, native-code libraries are only compiled if bytecode execution provides insufficient performance. Our common practice is to use bytecode except when running performance tests or when installing heavily used Ensemble executables.

### *4.4 Multi-threading*

Horus is a heavily threaded system. Every message received from the network causes a new thread to be created to handle it. In addition, every time a message is passed up in a protocol stack a new thread is also forked to handle that. However, optimizations are made so that in many cases the previous thread is recycled instead of forking an entirely new thread.

The issues related to threads in Ensemble are somewhat more subtle because its architecture is more flexible than Horus'. Although Ensemble is single threaded, there are a number of ways to introduce threads to the system. For instance, the C interface to Ensemble introduces C threads for the application, while Ensemble executes on in a single thread. In addition, each Ensemble protocol stack can optionally be configured to use a threaded implementation of the layering model (similar to Horus') instead of the default, unthreaded implementation which uses event queues.

However, one issue that has caused difficulty is that multi-threaded programs that mix ML and C are not as well supported as single-threaded ones. In the absence of threads, C code can call into ML code and vice-versa. With threads, these inter-

language interactions have to be more carefully managed because, in general, C code cannot call into ML code.

### 4.5 Sizes of executables

Ensemble and Horus executable binaries are approximately 880*K* and 400*K* bytes, respectively. These numbers are of course dependent on the platform, compiler, and level of optimization. These are the sizes of the stripped (i.e., without debugging symbol information) binaries for the default optimization level of the systems on SPARCStations running Solaris 2.5. From each system we eliminated a number of modules that provide functionality not present in the other system. For Ensemble, we give the size of the native code binaries. The bytecode binaries are 700*K* bytes. We believe the reason bytecode is not much smaller than native code is that Ocaml uses a 32-bit bytecodes.

### 4.6 Memory requirements

Ensemble and Horus have roughly similar memory requirements. At initialization, Ensemble processes with no protocol stacks use 91*K* bytes on the ML heap. In a similar configuration, Horus begins with 73*K* bytes on its heap. Each group joined by a process adds 7.8*K* bytes with Ensemble and 11*K* bytes with Horus. When the stacks become active (for a non-intensive application), the heap may grow to around 0.5*M* bytes for Ensemble and 1.5*M* bytes for Horus. The resident set sizes for these processes (again for a non-intensive application) are around 1.5*M* bytes for Ensemble and 2.5*M* bytes for Horus. We believe the additional space used for Horus is caused by the use of preallocated thread stacks.

### 4.7 Performance

Both Ensemble and Horus have very good performance. When transmitting 1*K* messages on SPARCstation 20's, both Ensemble and Horus are easily able to use the available bandwidth of a 10*M* bit Ethernet. In order to compare the efficiency of the common code paths in both systems, a good measurement is the application to application latency. This measurement is the average time to send and receive one message using Horus or Ensemble for the default protocol stacks. Note that while the 'abstract' protocols for the two systems are roughly similar (in both cases exactly one message is sent on the network), the protocol operations and header bytes are different. The bytes that are sent on the network by the two systems are different.

On SPARCstation 20's, Ensemble (compiled as native code) achieves a one-way latency of 595$\mu s$ and in a similar configuration Horus has 700$\mu s$. In this configuration, the overhead of the network is 355$\mu s$, so the latency induced by Horus is 345$\mu s$ and that of the Ensemble is 240$\mu s$. Garbage collection has a minimal impact on performance because Ensemble allocates very little memory on the heap in the normal cases for sending and receiving messages.

We believe the performance increase in Ensemble is primarily due to architectural

improvements. For instance, Ensemble compresses many of the headers that do not change between group reconfigurations. Another cause is that Ensemble protocol stacks are single-threaded by default, whereas Horus stacks are multi-threaded: profiling of Horus stacks has shown that as much as 20% of the execution time is spent in synchronization operations.

With the optimizations described by Hayden and van Renesse (1997), the overhead for Ensemble drops to 41$\mu s$ and the normal case protocol headers are further compressed down to 8 bytes. These same optimizations could be applied to Horus to achieve a similar speedup. However, many of the architectural improvements made to Ensemble would have to be replicated in Horus first, which would necessitate extensive rewriting.

### *4.8 Line counts*

Line counts are another useful characteristic for comparison even though comparisons of sizes of source code depend a large amount on factors such as the coding style of the programmers. See Table 2 for a number of different line count comparisons. All line counts are raw (comments have not been stripped) and include both implementation and interface files (i.e., .c and .h files for C, respectively).

- *Total lines* is the total number of lines in each system, including demonstration programs. This gives a sense of the overall sizes, but otherwise contains little information for useful comparison. Ensemble has considerably more demonstration programs than Horus. The bulk of the C code listed for Ensemble is for C and C++ interfaces, associated testing code, and an interface to the Electra CORBA-based replicated object system (Maffeis, 1995). No C code is actually needed to run Ensemble because the Ocaml UNIX library provides stubs for all needed system calls. However, Ensemble comes with its own set of UNIX stubs that can optionally be used to improve performance. These stubs amount to about 1000 lines of C code.
- *Core lines* is an estimate of the size of the core components in a stripped down system. This is what is needed to get standard configurations of each system running. Only minimal sets of protocol layers and none of the external language interfaces are included. For Horus, we only include the machine dependent code for a standard UNIX system. This measurement gives a more focussed picture of the code sizes of both systems.
- *Protocol lines* is the total number of lines of protocol layers. Both systems have around 55 layers. This measurement is very important because the protocol layers are the most complex parts of each system, and smaller layers tend to be easier to develop, debug, comprehend, verify, and maintain. The sizes of our protocol layers coded in ML are significantly smaller than those coded in C. This is discussed in more detail below.
- *Platform-dependent lines* is a count of lines of code used on a subset of the supported platforms. For Horus, this consists primarily of code for accessing system-dependent threads and messaging operations. Ensemble, which is

Table 2. *Comparison of lines of source code in Ensemble and Horus. See text for explanation.*

| measurement | Ensemble (ML) | Horus (C) |
|---|---|---|
| total lines | 45873 ( + 11000 C) | 140000 |
| core lines | ≈ 17000 | ≈ 35000 |
| protocol lines | 10692 | 79000 |
| average lines per protocol | 198 | 1519 |
| platform-dependent lines | 0 | 27020 |

Table 3. *Size comparisons of comparable Horus and Ensemble protocols.*

| C-layer | ML-layer(s) | C-lines | ML-lines | factor |
|---|---|---|---|---|
| Frag | Frag | 900 | 176 | 5.1 |
| Problem | Suspect | 1389 | 128 | 10.8 |
| Stable | Stable | 1639 | 318 | 5.2 |
| Credit | Credit | 2367 | 435 | 5.4 |
| Mbrshp | Inter:Intra:Leave:Merge:Elect | 5134 | 911 | 5.6 |

unthreaded by default, has effectively no such code (there is a little to work around idiosyncrasies of Windows 95 and NT), whereas Horus has a large amount. This is significant because platform-dependent code often creates software maintenance problems.

### *4.9 Sizes of protocol layers*

While the Horus and Ensemble infrastructures have diverged a good deal, they retain the same basic layered architecture. Many layers or collections of layers have direct analogues in both systems, thus allowing useful comparison of their number of lines. In general, the 'important' protocol layers in Ensemble are about a factor of 5 times smaller in lines than those in Horus. See Table 3 for a table of sizes of source code of comparable layers (or sets of layers) from both systems. Some of the differences in size can be attributed to differences in the language used, but some also to an overhaul in Ensemble of the general layering structure in Horus. For instance, Horus layers interact directly with thread and synchronization operations, whereas in Ensemble the infrastructure handles synchronization for all layers. If Ensemble were to be recoded in C, the sizes of the resulting layers would be significantly smaller than the Horus layers.

This said, we believe that the decrease in code size is also due in part to the use of ML as a programming language. There are several ways in which the programming

https://doi.org/10.1017/S0956796899003275 Published online by Cambridge University Press

language has had an impact. The first is that almost all data structures in Ensemble are managed automatically by ML. The explicit management of memory in Horus requires a large amount of code. The second is the better set of facilities in ML for abstraction, which encourages structural changes that result in decreased code size. Examples of this include the use of polymorphic abstract data types and higher order functions. The third is the ability to manipulate messages with standard language facilities such as pattern matching. Most communication systems use a special set of operators for manipulating messages because the contents must be linearized (marshalled) before transmission. Instead of treating messages as sequences of bytes, Ensemble uses ML data structures for all headers and linearizes them with an automatic marshalling facility provided by Ocaml. Not surprisingly, raising the abstraction level for Ensemble's core data type, messages, leads to significant reductions in code size.

### *4.10 Bugs*

The way we use ML prevents many kinds of bugs from occurring. For instance, when programmers add fields to the headers of protocol layers in Horus, they have to go through the protocols looking for all cases where a change needs to be made. In Ensemble, the headers are normal ML data structures, and this allows the compiler to detect and signal inconsistencies through its type checker. In addition, the ML marshaller handles converting headers into byte sequences for transmission on the network, so incompatibilities in byte ordering and word size are transparent to protocol layers in Ensemble. A user recently compiled and ran Ensemble on a machine with a 64-bit word size for the first time we are aware and encountered no problems. These kinds of problems were a constant concern in Horus. Bugs in Ensemble are generally bugs in the protocol logic and not memory management or message formatting errors.

### *4.11 Evolution*

Even though difficult to quantify, an important system characteristic is the ability to evolve. Both Horus and Ensemble are research systems that were intended in part to be toolkits to facilitate research in new protocol architectures. Our experience with Horus was that it did evolve a great deal for some time . However, it became increasingly difficult to make changes to the system. We believe this is because Horus became over-engineered generating a web of interdependencies. Often, these dependencies had to do with details of memory management or other issues that do not arise in ML programs. Ensemble has continued to evolve, often in dramatic ways. A detailed description of the evolution of one part is in section 6, but there are many other similar examples.

## 5 Messages

As a communication system, Ensemble's core data structures are messages. Most of the source code is involved with managing messages; most of the memory allocated

https://doi.org/10.1017/S0956796899003275 Published online by Cambridge University Press

is for messages; and most of the performance profile is involved in manipulating messages. Both architecturally and in terms of performance, communication systems are heavily dependent on the implementation of messages.

At the levels of the network and the application, messages take the form of flat sequences of bytes. This is because networks only transmit sequences of bytes, and because we wish to support application programs written in languages such as C, where messages are typically represented as sequences of bytes. Between the application and the network, protocols add information to messages in the form of *headers*. Headers are added at the sender by each layer and removed at the destination. For instance, headers are used to attach sequence numbers to messages in order to implement FIFO ordering protocols.

The approach we have taken in Ensemble is to use a low-level byte representation for the message payload and to use normal data structures for protocol headers. Thus, messages have two parts: payload and headers. The payload consists of a sequence of bytes, but the headers are regular data structures. The use of normal data structures for headers is an important design feature of Ensemble and a departure from many previous communication systems. This design means that the payloads (which most affect performance but which protocols typically do not access) have an efficient implementation, while the protocol headers (which protocol layers manipulate a great deal) benefit from extensive language support.

Our approach with header formats takes a somewhat non-standard view of how to format messages. In contrast with many other systems (Peterson *et al.*, 1993; Postel, 1981), the formats of headers for individual layers are not defined at the byte level. In addition, the headers of a stack of protocol layers is not the concatenation of the headers of the individual layers. Instead, the system is free to format protocol headers any way so long as all endpoints in a group use the same format. This gives Ensemble a great amount of flexibility in how headers are represented and in the methods for optimizing them. But it also raises questions about the drawbacks of this approach. The main drawback is that there are no simple, static formats to which programs must adhere with to communicate with an Ensemble processes. However, this would be difficult to achieve anyway because Ensemble embodies a more dynamic view of the world than most protocol architectures. Whereas the TCP protocol and its headers are not expected to change over the lifetime of a process, the Ensemble protocol stacks that an application uses do change dynamically. At any time, an application can request a protocol change that results in the process group switching to an entirely different set of protocols and headers. In addition, new protocols can be dynamically linked into Ensemble at run time. All of these forms of dynamicism argue against fixed header formats.

### *5.1 Protocol headers*

Many layered communication systems, such as Horus, view headers as extensions to the low-level representation of messages. In such systems, messages can be viewed as a 'stack' of bytes and the application and protocols use operations to push and pop bytes onto and off of messages. This design originated with the

```
/* total.c Message Headers */
struct to_header {
    uint8_t type;
    uint8_t flags;
    uint16_t dest;
    uint32_t token;
};

enum to_message_type {
    TO_TOKEN_REQUEST,
    TO_TOKEN,
    TO_DATA,
    TO_UNORDERED
};

/* Send a message. */
...
msg = horus_message_alloc(to_memory,
                          "TO_TOKEN_REQUEST");
horus_message_add(msg, 0, sizeof(*hdr),
                (void **) &hdr);
hdr->type = TO_TOKEN_REQUEST;
hdr->flags = 0;
hdr->dest = 0;
hdr->token = htonl(group->seqno);
err = horus_cast(group->below, 0, msg);
...

/* Receive a message. */
void handler(event *ev, horus_message *msg) {
  enum to_message_type type ;
  unsigned seqno ;
  ...
  switch (event.type) {
  case HORUS_CAST:
    err = horus_message_read_byte(msg,&type) ;
    if (!horus_err_ok(err)) ...
    switch (type) {
      case TO_TOKEN_REQUEST:
        err = horus_message_read_nlong(msg,&seqno) ;
        if (!horus_err_ok(err)) ...
        ...
        break ;
        ...
    }
    ...
  }
  ...
}
```

Fig. 3. Example of pushing and popping headers onto and off of a message in Horus.

X-Kernel (Peterson *et al.*, 1993), although UNIX STREAMS has a similar feature. There are a variety of reasons for this approach, such as the need to adhere to strict, standardized header formats and the expectation that low-level operations are needed to achieve high performance. Unfortunately, such designs have costs. These include the programming costs for having protocols directly handle byte ordering and word size incompatibilities between hosts. In addition, the use of low-level operations typically prevents a variety of high-level optimizations from being made to headers.

https://doi.org/10.1017/S0956796899003275 Published online by Cambridge University Press

```
(* total.ml Message Headers *)
type header =
 | TokenRequest of int
 | Token of (int * int)
 | Data of int
 | Unordered

(* Sending a message. *)
...
down (castEv name) (TokenRequest token) ;
...

(* Receiving a message. *)
let up_handler event msg = match (getType event), msg with
| Cast, TokenRequest token ->
    ...
| Cast, Unordered ->
    ...
| ...
```

Fig. 4. Example of headers in Ensemble.

In Ensemble, each protocol layer has a data type for the headers it puts on messages. A layer pushes a header onto a message by tupling its header with the headers of the layers above it. At the destination, the headers are unmarshalled from the messages and passed to the layers. Each layer in turn extracts its header from a tuple and passes the remaining headers up to the layer above. At the application, the last header is removed and all that remains is the message body, which is then passed to the application.

This still leaves the question of how the header object is linearized into a sequence of bytes at the bottom of the protocol stack so that it can be transmitted over the network along with the payload. Ensemble uses the Ocaml marshaller for this purpose. A marshaller is a function that takes a concrete data structure (embedded functions are not allowed) and linearizes it into a sequence of bytes from which a corresponding function can reconstruct a copy of the object. Marshallers typically transparently handle incompatibilities in byte ordering and word size. There are numerous standard marshalling formats such as XDR and ASN.1 (X.208, 1987). Ensemble uses the general-purpose marshaller in Ocaml, although it can easily support other marshallers.

By representing headers as regular data structures, protocols can leverage the same powerful language features, such as pattern matching and type checking, that are used for other data structures. This greatly simplifies the construction of protocols and eliminates a large number of programming errors. Not only does the programmer not have to handle complications from low-level details such as incompatible machine byte ordering and word sizes, but compilers can detect problems such as mismatched header types and cases where not all header combinations are handled. Using normal data structures gives the protocols a higher level of abstraction because many implementation details are hidden.

The use of a marshaller has the potential to add significant overheads to message size and processing which do not exist in an architecture where protocol headers

https://doi.org/10.1017/S0956796899003275 Published online by Cambridge University Press

are added using low-level operations. When using a marshaller, the headers are first constructed as data objects and are later linearized. The first step can be eliminated when writing the headers directly to the message.

However, we use a form of representation analysis to eliminate the use of the marshaller in the normal cases. The basic idea is to have protocol layers identify their normal-case headers to Ensemble. Ensemble uses this information to identify normal-case messages and to bypass the general-purpose marshaller with one that has been optimized for simple cases. The optimized version effectively eliminates the latency and message-size overhead of the general-purpose marshaller. This optimization is transparent to the protocols and requires no modification of the layers other than to identify their normal case headers during initialization (see Hayden and van Renesse (1997) for further details). As with automatic garbage collection, marshalling was useful because it let us focus on just the critical cases by automating the rest.

#### *5.1.1 Special purpose marshallers*

Although Ensemble currently uses the Ocaml general-purpose marshaller, we are experimenting with using special purpose marshallers compiled from type information provided by the Ocaml compiler. The normal Ocaml marshaller takes an arbitrary ML data structure and uses tags in the data representation to marshal it. A marshaller specialized to the actual data type of a message can achieve a more compact representation and smaller marshalling/unmarshalling times than the general purpose marshaller, but the main benefit is that it would be better able to detect malformed messages. This is important for security in settings where an intruder may attempt to crash other processes by sending so-called *poison-pill* messages that are designed to violate the typing expectations of the protocols and cause run-time type errors (which usually crash the process).

### *5.2 Message payloads*

Whereas protocol headers provide many opportunities to make use of features of ML, the application payload portion of messages raises a series of issues. This is largely because the nature of message payloads requires that they be represented as low-level sequences of bytes. One normally thinks of ML as a language best suited for manipulating high-level objects, and sequences of bytes fall outside of the domain where many features of ML can help. For instance, sequences of bytes in the payload often represent some high-level object, but type checkers are usually not able to capture this structure in useful ways. Thus, the question arises of whether ML is a good language for doing systems development where low-level objects often occur and where language support for them is important. Indeed, a major reason Ensemble benefits from the use of ML is that we have succeeded in abstracting much of the system at a high enough level that features of ML pay off. It is primarily in message payloads that Ensemble confronts issues associated with low-level objects. However, we feel that it is typical of many domains that most of the problem can be abstracted above low-level issues.

A message payload implementation must support a variety of operations, including allocation, release, subset (creating a new message from a subsequence of the bytes in another) and catenation. The subset and catenation operations are needed mostly for fragmentation-reassembly and message packing protocols. For instance, a fragmentation-reassembly protocol needs to be able to break a large message into smaller messages that fit the maximum message size supported by the network and reassemble it at the destination. All these operations should be efficient for messages of sizes ranging from 0 bytes to at least 10*K* bytes. They should not cause additional allocation for the body of the message nor should they copy the contents.

Message payloads in Ensemble are represented as arrays of records called **iovec**s (based on the UNIX data structure of this name). An **iovec** contains a pointer to a string, an integer offset, and an integer length. The offset specifies where the **iovec**'s body begins in the string and the length gives the number of bytes of data. Both the string and the other fields of the **iovec** are treated as read-only. **Iovec**s are exported to the rest of the system as an opaque, abstract data type. A subset of an **iovec** is created by allocating a new **iovec** record with the same string as the original but with different offset and length. Catenation is done by catenating arrays of **iovec**s. Thus, neither subset nor catenation copy the contents.

Some recent work has focused on introducing support in ML implementations for low-level data structures such as untagged word arrays (Tarditi *et al*., 1996). Such support is justified in part by the claim that it is needed in order to do real low-level systems work in ML. While this may be true for programs that interface directly with device drivers (for instance), Ensemble interacts with the network through system call stubs written in C, and it has not suffered from the absence of untagged word arrays. It would have been nice to have been able to implement Ensemble entirely in ML without these stubs, but the addition of less than 1000 lines of simple C code is a relatively insignificant portion of the system. Our experience with Ensemble has shown that the **string** core data type wrapped in **iovec**s with support from C system call stubs is sufficient at least for the needs of communication systems development. This not to say that untagged word arrays would not improve performance and/or memory usage, only that they are not a prerequisite to doing high-performance systems work.

## 6 Buffer management

Although ML strings wrapped with **iovec** records are sufficient for efficiently manipulating message payloads, a variety of memory management issues arise regarding how **iovec** strings are allocated and managed. The exact issues are involved with details of the Ocaml garbage collector, but the general lesson we learned was that garbage collectors may not be the best mechanism for managing data structures with a major impact on performance, such as messages. Some garbage collection strategies can cause unnecessary copying of data, bad fragmentation of memory, and slow recovery of memory. In the end, these problems drove us to explicitly manage message buffers, even though the rest of the system still benefits from automatic memory management.

### *6.1 First implementation*

In our initial design, Ensemble allocated a new string prior to receiving a message from the network. This caused a variety of problems. First, because the length of a message received from the network is not known in advance, a string of the maximum transmission length had be allocated. For instance, the **recv()** system call in the UNIX BSD socket interface must be passed a buffer with sufficient space to contain the largest expected message size. This size can be up to 64*K* bytes, but for a number of reasons Ensemble typically uses messages of at most 10*K* bytes. Allocating a 10*K* byte block every time a message is received wastes a great deal of memory when the actual size turns out to be much smaller. In the case of a 100 byte message, 99% of the 10*K* byte space is wasted from internal fragmentation. Internal fragmentation refers to unused memory space allocated within an object and can result in a large waste of memory, as in our case. There are a variety of potential solutions to this problem. One option is to copy the message out of the buffer into a new string of the appropriate size. However, this causes a copy for each message, and external fragmentation (unused space outside of objects) is still a problem because the Ocaml garbage collector (as with many non-copying collectors) does a poor job of preventing fragmentation when there are large blocks of varying sizes (Wilson *et al.*, 1995).

### *6.2 Using large message buffers*

Both sorts of fragmentation are avoided in Ensemble by using very large strings for allocating **iovec**s. These strings are called *segments* and are typically 256*K* bytes long. Segments are managed by **msgbuf**s, which consist of a current segment and offset. Allocation from **msgbuf**s is done by creating an **iovec** record with the **msgbuf**'s segment and offset, and the desired length. The offset of the **msgbuf** is then advanced. If there is no longer enough space left to allocate the maximum size block from the **msgbuf**, a new segment is allocated and the offset is reset to zero. Allocation and release of **iovec**s from **msgbuf**s are inexpensive operations. Allocation usually consists of just advancing the **msgbuf** offset. Deallocation is done by the garbage collector once for each segment when there are no more references to a segment. The use of large segments has the potential drawback that a segment can only be released after the release of the last message using it. However, practice has found that this is not a problem as messages tend to have similar expected lifetimes.

The problem that arises with this design is that under even moderate loads much of the execution time (more than 25%) is used by the garbage collector in order to recover segments. Because we do a good job of avoiding other allocation, there is very little dead data to collect other than the (albeit large) segments, so these collections are inefficient in terms of the number of objects release per collection. The cost is not from the actual allocation or deallocation operations on the segments (and the collector does not copy the segments), but from the garbage collector scanning all live objects in the heap in order to deallocate the relatively small number of free segments. Ocaml could be configured to wait longer between garbage collections, but this causes a lot of memory to be wasted. So we changed our approach again.

https://doi.org/10.1017/S0956796899003275 Published online by Cambridge University Press

### 6.3 Reference counted management

We decided to add explicit reference counting to buffers. Each segment has an associated reference count that keeps track of the number of references to the buffer. When the reference count drops to zero, the segment is returned to a free list maintained by Ensemble. This form of reference counting is simple to implement because the objects being managed do not contain references to other objects. It eliminates our problems with the garbage collector because memory allocated for message payloads is rapidly recovered without requiring a garbage collection. The Ocaml garbage collector is only triggered by allocation on its heap, so when the apparent allocation rate decreases, the rate of garbage collections does also.

The use of reference counting adds some programming cost because protocols must correctly update the reference counts. We found that this was easy to do because the reference count operations are only needed when a protocol layer releases or buffers a message, and these operations are simple to recognize. The computational overhead of maintaining the reference counts is quite small because the compiler inlines the reference count operations at the call-site (we describe this in detail in section 7). In addition, operations for managing the segment free list are efficient because the cost for each segment is amortized over all of the messages allocated from it.

The average time to allocate messages is graphed in figure 5 for message sizes ranging from 4 bytes to $8K$. The measurements were taken on a 200 Mhz Intel Pentium Pro processor and were made by first growing the heap to a typical size for Ensemble and then allocating and releasing 50000 objects. The x-axis denotes the size of messages being allocated. The y-axis denotes the average time (in microseconds) to allocate and release one message. We do not include the time for the system call to receive the message. Note that both axis have logarithmic scales. The four lines correspond to: (a) allocating 10K byte strings for every message, (b) using a fixed 10K byte string to receive the message and then copying into a string of the correct size, (c) **msgbuf**s without reference counts, and (d) **msgbuf**s with reference counts. Option (a) is almost uniformly the worst. For 20 byte or smaller messages, option (b) performs best. For larger sized messages, (c) begins to perform better than (b) because the cost of copying starts to dominate the cost of garbage collection. (b) climbs significantly at 1024 bytes because this is the lower threshold for allocating objects on the major heap (allocation/freeing on the Ocaml major heap is significantly more expensive than the minor heap). (c) also climbs with message size because the **msgbuf**s are filled up more rapidly, which in turn increases the rate of major garbage collections. Option (d), however, is always close to the others and maintains low latencies throughout the range of message sizes (although in this test (b) has better performance than (d) for messages of less than 20 bytes, in the actual use in Ensemble (b) and (d) exhibit equivalent performance for these message sizes). This is because the garbage collector is rarely being activated.

Reference counts introduce the concern that they can cause both memory faults and memory leaks due to programmer errors. However, these problems can be prevented in Ensemble by enabling an option prior to compiling that causes segment

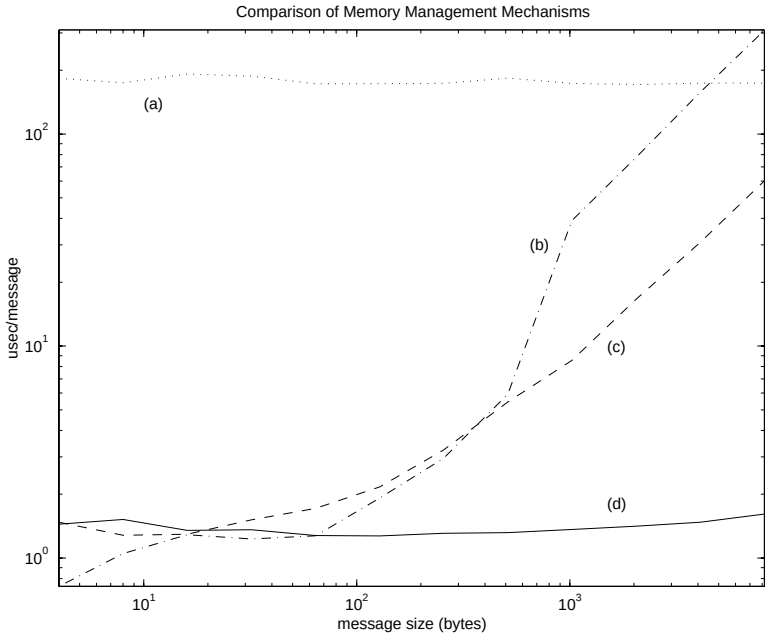


Fig. 5. Comparison of the performance of management mechanisms used for **iovec**s. See text for an explanation.

reference counts to be checked before accessing the segment. This slows execution somewhat, but prevents memory errors. The opposite problem, memory leakage, can occur if reference counts are occasionally not decremented to zero, causing segments to never be released. We addressed this problem by using weak pointers (Hayes, 1992) to detect when the reference count object (which wraps the segment) has no further references. When an object with non-positive reference count is accessed or when an object's reference count is not decremented to zero when the weak pointer is released, the problem is signaled to the user (and in the latter case the segment is recovered to prevent a memory leak: see figure 6).

Both ISIS and Horus had similar problems. Messages are no less crucial data structures there than they are in Ensemble. The use of the system-provided memory allocation and release operations (**malloc()** and **free()**, respectively) were insufficient for managing messages, and both ISIS and Horus ended up developing their own sub-systems for managing memory associated with messages. These message sub-systems involved complex, multi-level, reference-counted data structures with special-purpose free lists containing preallocated and pre-formatted objects. The result was that message management was both more complex than in Ensemble and had worse performance.

In summary, the memory management facilities for Ocaml turned out to be insufficient for Ensemble and we had to add our own support to the system. It is important to be careful in how one views this. One could say that this is a failure of a garbage collected language because the garbage collector was not powerful enough

https://doi.org/10.1017/S0956796899003275 Published online by Cambridge University Press

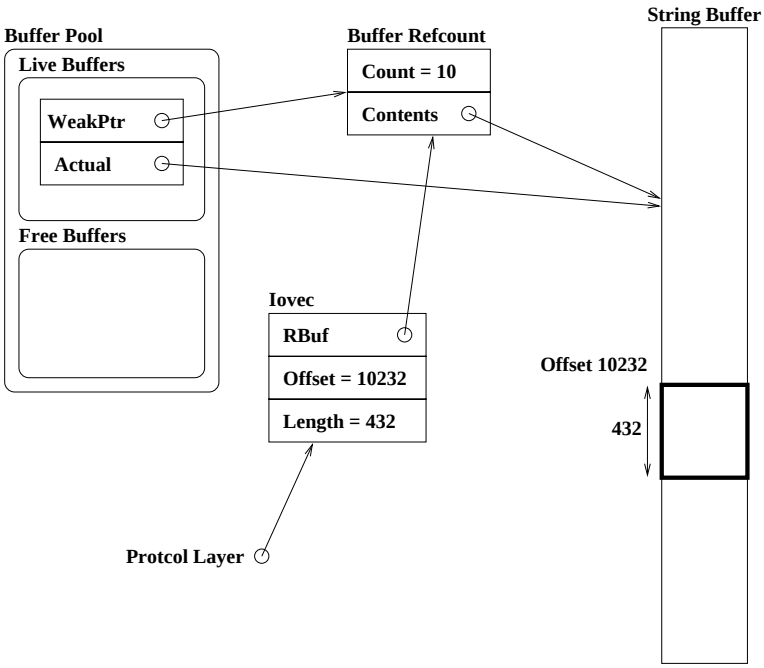


Fig. 6. Depiction of the revised **iovec** structure. A protocol layer is given a pointer to an **iovec** record. The **iovec** contains a pointer to a **refcount** record. The **refcount** contains a reference count and a pointer to the string buffer. The **iovec** record contains the integer offset and length of data in the **Refcount** string. The buffer pool contains a weak reference to the **Refcount** record and a pointer to the string buffer.

to handle everything. However, it is a characteristic of systems style work that there are often a small set of data structures which require very careful management to achieve high performance. This was the case in Horus and ISIS, and it turns out that Ensemble is not any different in this respect. Ensemble's special management of messages highlights the usefulness of automatic garbage collection: the garbage collector handles the vast majority of memory management, allowing us to focus on the cases where specialized management is required. Whereas all data structures are explicitly managed in Horus and ISIS, in Ensemble only one is treated in this manner.

## 7 Inlining

Communications systems and other programs with a strong systems flavor often have multiple levels of abstraction barriers that must be crossed to manipulate data structures, even though the abstraction barriers often hide relatively small pieces of code. Ensemble exhibits this structure because of its layering and extensive use of modules. The abstraction barriers are useful because they increase modularity. The problem, of course, is that when there are lots of abstraction barriers to be crossed in doing inexpensive manipulations, the cost of the abstraction barriers (in the form of function call overhead) can be larger than the cost of the manipulations themselves.

Programmers in other languages such as C are familiar with this problem. They solve it by using macros or an inlining mechanism provided by the compiler. However, this often requires that the programmer annotate which functions are to be inlined. Ocaml also provides inlining support that eliminates the cost of these abstraction barriers, but without requiring annotation by the programmer. Inlining involves copying the body of a function to the sites at which the function is called, thereby eliminating the overhead of a function call. This is done at the potential cost of causing growth in the size of the compiled code, but we have not had any problems from the size exploding because the Ocaml compiler has a variety of heuristics to prevent excessive growth.

To see how all of this works, we present an operation in Ensemble, that of releasing the contents of an Ensemble **Event** data structure (done at least twice for every message), and show how inlining eliminates the potentially costly abstraction barriers. In addition to the call site, there are four modules in this example. The call to the function **Event.free** results in turn to calls to **Iovec_array.free**, **Iovec.free**, and **Refcount.decr**. Each step adds one small part to the overall operation, such as dereferencing a record's field and calling another function on that field.

Through all the modules, string arguments are passed for use in debugging. These arguments are called **debug** and the string passed in this example is **"FIFO"**. With this debugging information, all of the modules described here can be compiled to emit detailed traces. However, these debugging arguments are ignored in the normal code for the modules presented below.

```
(* call site *)
 ...
 Event.free "FIFO" ev
 ...

(* Event module *)

type t = {
  ty            : typ ;
  origin        : rank ;
  ranks         : rank list ;
  ack           : acknowledgement ;
  iov           : Iovec_array.t ;
  extend        : field list
}

(* Call the Iovec_array.free function on the iov
 * field.  Pass on the debug string unchanged.
 *)
let free debug ev =
  Iovec_array.free debug ev.iov

(* Iovec_array module *)

type t = Iovec.t array

(* Call the Iovec.free function on each entry in
 * the array.  Pass on the debug string unchanged.
 *)
let free debug ia =
  for i = 0 to pred (Array.length ia) do
    Iovec.free debug ia.(i)
  done
```

https://doi.org/10.1017/S0956796899003275 Published online by Cambridge University Press

```
(* Iovec module *)

type t = {
  rbuf : rbuf ;
  ofs : ofs ;
  len : len
}

(* Call the Refcount.decr function on the rbuf
 * field of the record.  Pass on the debug string
 * unchanged.
 *)
let free debug i =
  Refcount.decr debug i.rbuf

(* Refcount module *)

type 'a t = {
  mutable count : int ;
  obj : 'a ;
  mutable debug : (string * string) list
}

(* Decrement the reference count.  This version
 * ignores the debug argument.
 *)
let decr debug r =
  r.count <- r.count - 1
```

When compiled, all four levels of function calls are inlined at the caller. This happens even though all the module interfaces export abstract data types. The cost of crossing all of the abstraction barriers has been eliminated by inlining across modules. The call to **Event.free** is inlined by Ocaml like this:

```
let iova = ev.iov in
let lo = 0 in
let len = Array.length iova in
let hi = len - 1 in
for i = lo to hi do
  let iov = iova.(i) in
  let refcount = iov.rbuf in
  refcount.count <- refcount.count - 1
done
```

Note that because of abstract module interfaces, it is not possible for a programmer to write code that directly accesses the data structures as in the code generated after the inlining. Also, there was no cost at run time for passing debugging strings to the functions: the inlining exposed to the compiler the fact that the debugging string was not being used in the modules and so it was eliminated. This is the resulting assembly code (for the Intel 386 instruction set) that is generated for the original call to **Event.free** (annotated with the corresponding ML code from above):

```
        movl    16(%eax), %ecx          # let iova = ev.iova
        movl    $1, %eax                # let lo = 0
        movl    -4(%ecx), %ebx          # let len = Array.length iova in
        shrl    $9, %ebx                #   contd.
        orl     $1, %ebx                #   contd.
        addl    $-2, %ebx               # let hi = len - 1 in
.L105:
        cmpl    %ebx, %eax              # for-loop termination
        jg      .L104                   # escape if done
```

https://doi.org/10.1017/S0956796899003275 Published online by Cambridge University Press

```
movl    -2(%ecx, %eax, 2), %edi # let iov = iova.(i) in
movl    %edx, %esi              #   contd.
movl    (%edi), %esi            # let rc = iov.rbuf in
addl    $-2, (%esi)             # rc.count <- rc.count - 1
addl    $2, %eax                # i <- i + 1
jmp     .L105
```

There is some room for further minor optimizations in the resulting code (mainly restructuring to eliminate one of the branch instructions in the loop), but the compiler has done a very good job of eliminating the abstraction barriers from the resulting code. Achieving similarly optimized code in C while maintaining opaque abstraction barriers would not be easy; in Ocaml it takes no additional work on the programmer's part.

## 8 Conclusion

Our experience in developing Ensemble with Ocaml was very good. Strong static type checking was extremely useful. Most of the other language features had a set of benefits in development along with drawbacks to performance. Because of this it was important to make careful use of features in order to maintain efficiency in the final implementation.

Throughout this paper, we have attempted to draw general conclusions where possible in order to try to answer the question of when it makes sense to use an advanced language such as ML for developing systems-style software. Perhaps the first consideration is the willingness of a group of people to work with a new programming language. In particular, ML has a number of aspects, such as type checking, that many programmers find discomforting at the start. Acquiring enough inertia to move beyond these sorts of issues can be difficult, even in open-minded research settings.

Many of the problems that existed in earlier implementations of ML have been overcome in newer systems, leaving a number of advantages to building systems in ML. Strong static type checking, polymorphic functions, formal module systems, and automatic garbage collection are some of these. For communication systems, automated marshalling is also very useful. However, a number of improvements could be made to Ocaml and other ML implementations to better support systems programming.

- More general forms of memory management, including mixing different mechanisms (for instance, mark-and-sweep and reference-counting), and also giving the programmer more control when needed. We are not aware of ML implementations that provide such support (although there are some for C (Wilson *et al.*, 1995)). Instead of adding more powerful general-purpose memory management, it may be sufficient to provide ML libraries for efficient buffer management, combined with support for file and network operations.
- Compacting the memory heap for mark-and-sweep garbage collection. This is necessary for efficient memory usage in long-running processes (but of course is not an issue with copying garbage collection).

https://doi.org/10.1017/S0956796899003275 Published online by Cambridge University Press

- Although we did not need untagged data structures in order to achieve high levels of performance, such support would likely improve performance and simplify the use of external modules written in languages such as C.
- Interfacing well with C-based threading libraries. In interfacing Ensemble with multi-threaded applications in C, we ran into difficulties because C-threads were not supported by Ocaml[3]. This meant that Ensemble had to run as a separate thread which communicates with the application threads via shared message queues. All of these threading issues had to be handled in C. We are not aware of ML implementations that support C threads.
- Better inlining. While doing a good job with inlining, the Ocaml compiler does not inline higher order iterators for arrays and lists, and this causes unnecessary memory allocation for closures.
- Support for protecting the garbage-collected heap from buggy C libraries to make debugging of hybrid systems easier.

It is important to be aware that ML/Ocaml is not a panacea. There are places where the support provided by the language may be inadequate, and in such cases it is necessary to regain control from the language, as we did with the management of message buffers in Ensemble. Other kinds of operations, such as interfacing to hardware or low-level device drivers, may also require writing some stub code in C. Such stubs are often not difficult to write, but they are a consideration if a system requires a large number of them.

For systems style work in an advanced programming language, the challenge is often to leverage the advantages of the language. The low-level details that appear in systems work tend to eat away at these advantages. Thus it is crucial that the system be designed to keep as much as possible at a high level of abstraction. Support for inlining and other features can help in achieving this by eliminating much of the cost of abstraction barriers. If the low-level details can be limited to small parts of the system then there are many reasons to expect benefits from the use of advanced languages.

## Acknowledgements

Greg Morrisett, Ken Birman, Robert Constable, Robert Harper, Peter Lee, Robbert van Renesse, and Samuel Weber contributed valuable comments on earlier versions of this paper. This paper has also benefitted from the referees' comments. Contributors to the implementation of Ensemble include Tim Clark, Chris Driggett, Pedro Fheas, Roy Friedman, Takako Hickey, Ohad Rodeh, Robbert van Renesse, Alexey Vaysburd, Werner Vogels, and Zhen Xiao. Robbert van Renesse was the primary implementor of the Horus system, from which Ensemble adopted many ideas. He also contributed information about the effect of C on its development. Ken Birman contributed information about ISIS. Alexey Vaysburd implemented the C and C++

[3] Ocaml currently supports multi-threading for bytecode executables. However, the threads all share a single C thread rather than run on their own threads.

interfaces described here. The design of the Ensemble message buffers arose out of discussions with Jason Hickey and Werner Vogels. Much of this work would not have been possible without the great job done by the Ocaml developers.

## References

Armstrong, J., Williams, M., Wikstrom, C. and Virding, R. (1996) *Concurrent Programming in Erlang*. Prentice Hall.

Atkinson, R. (1995) *Security Architecture for the Internet Protocol*. RFC 1825.

Biagioni, E. (1994) A structured TCP in Standard ML. *Proc. ACM Symp. on Communications Architectures & Protocols*.

Biagioni, E., Harper, R., Lee, P. and Milnes, B. G. (1994) Signatures for a network protocol stack: A systems application for Standard ML. *Proc. ACM Conf. on Lisp and Functional Programming*.

Birman, K. P. (1996). *Building Secure and Reliable Network Applications*. Manning Publishing Co./Prentice Hall.

Birman, K. P. and Joseph, T. A. (1987) Exploiting virtual synchrony in distributed systems. *Proc. 11th ACM Symp. on Operating Systems Principles*, pp. 123–138.

Birman, K. P. and van Renesse, R. (1994) *Reliable distributed computing with the Isis Toolkit*. IEEE Press.

Constable, R. L. *et al.* . (1986) *Implementing Mathematics in the NuPRL Proof Development System*. Prentice–Hall.

Harlequin (1996) *The MLWorks user guide*. The Harlequin Group, Cambridge.

Hausman, B. (1994) Turbo Erlang: Approaching the speed of C. In Tick, E. and Succi, G. (eds.), *Implementations of Logic Programming Systems*, pp. 119–135. Kluwer Academic.

Hayden, M. (1998) *The Ensemble System*. PhD thesis, Cornell University.

Hayden, M. and van Renesse, R. (1997) Optimizing layered communication protocols. *Proc. Symp. on High Performance Distributed Computing*.

Hayes, B. (1992) Finalization in the garbage collector interface. In Bekkers, Y. and Cohen, J. (eds.), *International Workshop on Memory Management: Lecture Notes in Computer Science 637*, pp. 277–298. Springer-Verlag.

Kreitz, C. (1997) *Formal reasoning about communication systems I: Embedding ML into type theory*. Technical Report TR97-1637, Cornell University.

Leroy, X. (1997) *The Objective Caml system release 1.05*. INRIA, France.

MacQueen, D. (1993) Reflections on Standard ML. In Lauer, P. E. (ed.), *Functional Programming, Concurrency, Simulation and Automated Reasoning: Lecture Notes in Computer Science 693*, pp. 32–46. Springer-Verlag.

Maffeis, S. (1995) Adding group communication and fault-tolerance to CORBA. *Proc. USENIX Conference on Object-oriented Technologies*. Monterey, CA: USENIX.

Marlow, S. and Wadler, P. (1997) A practical subtyping system for erlang. *Proc. ACM SIGPLAN International Conference on Functional Programming*, pp. 136–149.

Peterson, L. L., Hutchinson, N., O'Malley, S. and Abbott, M. (1993) RPC in the x-Kernel: Evaluating new design techniques. *Proc. 14th ACM Symp. on Operating Systems Principles*, pp. 91–101.

Postel, J. (1981) *Transmission Control Protocol*. RFC 793.

Schneider, F. B. (1990) Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, **22**(4), 299–319.

Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R. and Lee, P. (1996) TIL: A

type-directed optimizing compiler for ML. *Proc. SIGPLAN Conference on Programming Language Design and Implementation.*

van Renesse, R., Birman, K. P. and Maffeis, S. (1996) Horus: A flexible group communication system. *Commun. ACM*, **39**(4), 76–83.

Weis, P. and Leroy, X. (1993) *Le language Caml.* Paris: InterEditions.

Wilson, P. R., Johnstone, M. S., Neely, M. and Boles, D. (1995) Dynamic storage allocation: A survey and critical review. *Proc. International Workshop on Memory Management*.

X.208, CCITT Recommendation (1987) *Specification of Abstract Syntax Notation One (ASN.1)*.