The Horus and Ensemble Projects

Accomplishments and Limitations

Ken Birman, Bob Constable, Mark Hayden, Jason Hickey, Christoph Kreitz, Robbert van Renesse, Ohad Rodeh and Werner Vogels[1]

Abstract– The Horus and Ensemble efforts culminated a multi-year Cornell research program in process group communication used for fault-tolerance, security and adaptation. Our intent was to understand the degree to which a single system could offer flexibility and yet maintain high performance, to explore the integration of fault-tolerance with security and real-time mechanisms, and to increase trustworthiness of our solutions by applying formal methods. Here, we summarize the accomplishments of the effort and evaluate the successes and failures of the approach.
Index Terms– Reliable Multicast, Fault Tolerance, Distributed Systems Security, Distributed computing, Automated Verification, Real-Time Cluster Computing.

I. A BRIEF HISTORY OF FAULT-TOLERANT PROCESS GROUP MECHANISMS FOR RELIABLE DISTRIBUTED COMPUTING

Prior to 1985, distributed computing was dominated by the Internet, and the Internet (in turn) was dominated by point-to-point communication mechanisms providing best-effort reliability. Although certain services provided forms of replication (for example, the network news program, the DNS, "yellow pages" (later renamed NIS) and the Xerox Clearinghouse system), application-level support for replicated data was lacking, and even replicated services were typically hardwired to support predetermined sets of clients. Applications built over the Internet employed TCP, UDP and FTP, treating the implementations of these protocols and the mechanisms supporting Internet routing and DNS address resolution as opaque components of the network itself.

During a period from 1983-1985, Cheriton and Zwaenapoel at Stanford extended the basic IP communication suite to support what they called distributed process groups [CZ85]. They used these groups in support of application-initiated broadcast, and proposed a number of group-based services which used broadcast to provide some form of parallelism or accelerated response. Inspired by their work, our fault-tolerant communications effort at Cornell proposed a distributed computing model based on process groups, but extended by the introduction of formal semantics for error handling [BJ87].

The approach that we proposed (soon adopted by several other research groups) became known as reliable process group computing, or virtually synchronous process group computing. In essence, a virtually synchronous process group provides automatically managed membership for application programs, which are permitted to join and leave groups and are informed of membership changes by upcalls. Also provided are multicast interfaces supporting ordered message delivery, and a means of initializing new members when they join the group - we call this the state transfer problem. To ensure that the solution is powerful enough to permit replication of data, membership change is synchronized with respect to multicast sending and delivery, and state transfer is implemented to appear atomic with respect to membership change.

Figure 1 illustrates this model graphically, where time advances from left to right. We see a process group within which multicasts are being used to update the states of members. When new members join, state transfer is shown by a thick arrow. Notice that arrows in the figure are drawn to suggest that events occur synchronously - as if all members that experience the same event, experience it at the same moment in time. Virtual synchrony is "virtual" in the sense that without using a synchronized distributed real-time clock, a running program will be unable to distinguish an actual execution from some closely synchronous one, such as the one in the figure. In reality, however, the events in a virtually synchronous execution may be highly asynchronous. The major benefit of this approach is that by relaxing synchronization in ways that participating processes can't detect, we are able to provide very high performance. Yet the execution model is intuitively simple and makes it easy for application developers to implement very complex, fault-tolerant, distributed services.

The virtual synchrony model was rapidly adopted by research and industry groups world-wide [Bir99]. Some successes associated with our work on the Isis Toolkit include the overhead display systems that show stock price quotes and transactions on the floor of the New York Stock Exchange [Gla98], the entire communications architecture of the Swiss Exchange [PS97], the console clustering architecture used in a new generation of air traffic control technology recently rolled out in France [Bir99], the control subsystem of several major VLSI fabrication plants (AMD, Seimens, Texas-Instruments), and a number of mobile telephony products.

[1] Dept. of Computer Science, Cornell University, Ithaca NY 14853. ken@cs.cornell.edu. Hayden is at the Systems Research Center of Compaq Corporation, Palo Alto CA. This work is supported in part by ARPA/ONR grant N00014-92-J-1866, ARPA/RADC grants F30602-96-1-0317 and F30602-98-2-0198, and NSF grant EIA 97-03470

Military uses of the technology included an intelligence monitoring and reporting technology implemented by NSA, a prototype for the next generation of the AEGIS Naval radar and communications system (called HiperD, this was the basis of the SC-21 standard, which in turn is the basis for DD-21, an important military communications standard expected to have wide-ranging impact during the coming decades) and certain applications associated with Ballistic Missile Command and Control.

These successes can be traced to the ability of the model to support replicated data, to provide high availability (in contrast to database replication methods, which guarantee recoverability but at the cost of sometimes needing to wait for a failed system to recover before the system can resume providing services - a source of potentially long outages), and to support load-balancing within small cluster-styled servers. Yet while these were important accomplishments, virtual synchrony also presented many drawbacks [Bir99].

Early implementations of the model were monolithic and relatively inflexible: A system like Isis was built from floor to ceiling with one form of communication in mind, and to the degree that one wished to turn features on or off, the technology tended to come with huge numbers of specialized interfaces that the programmer needed to learn and use selectively. For example:

- Isis supported several forms of message ordering. The more costly forms of ordering were also the easiest to use, but the performance hit was considerable. Developers were often forced to tune the choice of message ordering to obtain good performance.

- When an application supports multiple, overlapping process groups, there are many options for the way that events should occur within processes belonging to more than one group. Control over these options was required because no simple default emerged.

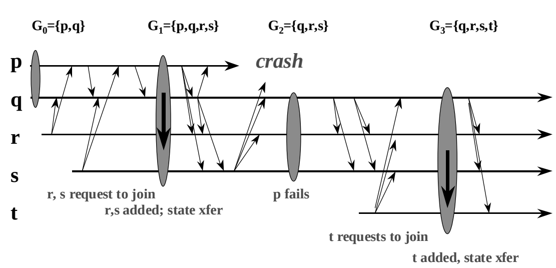

Figure 1: Virtual synchrony model, showing executions of five processes (time advances left to right)

- Some applications required that messages be encrypted but this was a significant cost in Isis, so the feature was only enabled as needed.

Moreover, at the time Isis was developed, the state of the art for object oriented design was very primitive. CORBA and DCOM/OLE had yet to be introduced, and even RPC had yet to be standardized. As a consequence, early systems like Isis, which used object-oriented designs and concurrent styles of programming were forced to introduce their own solutions. For example, Isis had a widely used threads implementation at the time that CMU first began work on cthreads and pthreads, the threads library that ultimately became standard in Linux. Over time, developments in these areas made Isis less and less compatible with commercial trends.

An additional side-effect of having large numbers of very demanding users was that Isis became more and more complex over time, and it became harder and harder to convince ourselves that the technology itself was free of bugs. Such developments created the concern that process group communication merely invites users to place all their eggs in one basket, and that the basket itself could break, exposing the entire application to mishap. Isis was relatively robust, but it took years to achieve continuous availability with the technology, and when an Isis protocol error surfaced, it could easily bring down an entire distributed system.

II. GOALS OF THE HORUS AND ENSEMBLE PROJECTS

Starting in 1990, we began work on the Horus system [RBM96] to try and overcome these first-generation considerations; Ensemble was subsequently started as a sibling of Horus in 1996. The basic idea underlying both projects is to support group communication using a single generic architectural framework within which the basic group communication interfaces are treated separately from their implementation. One can then plug in an implementation matching the specific needs of the application. To maximize flexibility, each group end-point instantiates a stack of what we call micro-protocols. The developer arranges for the stack used in support of a given group to provide precisely the properties desired from the group. Each micro-protocol layer handles some small aspect of these guarantees. Figure 2 illustrates this idea. Each process in a process group is supported by an underlying protocol stack; the stacks for the various members are identical, but the stacks used in different groups might be very different from one-another.

For example, one layer might overcome message loss by numbering each message and retransmitting lost messages in response to NAK's. The layer would have an outgoing and an incoming side. Outgoing messages would have a header added to them containing a sequence number, and would be stored pending garbage collection. Incoming messages would be examined to make sure they are in sequence. If not, a NAK message is sent back to the original sender of the message soliciting a retransmission. A separate layer detects when all copies have been delivered (a property called stability) and triggers garbage collection for stable messages.

A trivial example of the opportunity afforded by such an architecture is that the NAK layer might not be needed in some situations - namely, those in which the network doesn't lose messages. At runtime, based on the environment, the Horus system was capable of assisting the user in configuring a stack to provide reliability, inserting a NAK layer if necessary and omitting it if not.

Of course, useful protocol stacks often contain many layers and the kinds of layers one might selectively omit tend to be much more costly and less often needed than NAK. For example, security is often enabled selectively in Horus, the choice of protocol suite implementing virtual synchrony is often of importance (different protocols can give very different performance, and some are much better than others on specific hardware platforms), and different ways of doing multicast ordering are favored under different conditions. Without belaboring the point, the approach provides enough flexibility so that application designers with very different goals can potentially agree on the sharing of a common infrastructure, within which their commonality is captured by layers that they share, and their differences reflected by layers built specifically for their special needs.

Moreover, Horus can potentially support execution models very different from virtual synchrony. Our early hope was that the protocol interfaces could be offered as a standard, and that implementations of such protocols as SRM and RMTP-II, two widely popular scalable protocols with weaker reliability models, might be developed to run on the same platform. Unfortunately, in 1999 as this paper was being written, discussion of possible standards along these lines were still advancing very slowly within the IETF and OMG, two organizations that have shown an interest in developing such standards.

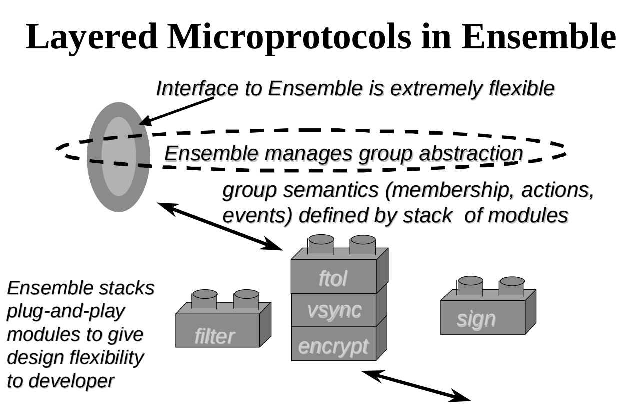

Figure 2: Ensemble and Horus used layered architectures.

The Horus effort did more than to simply support a layered stackable architecture. We also wanted to demonstrate that the performance of our architecture could be as good or better than that of a conventional monolithic architecture. We sought to provide real-time features, in response to a requirement coming from the Naval AEGIS application, where there was a need for cluster-style servers able to guarantee real-time event response even under stress (the application involved weapons targeting using radar tracks and had very tight time constants associated with acceptable responses). And whereas Isis was initially focused on securing its own abstractions, Horus was designed to offer security services on behalf of application developers who needed security key infrastructures for purposes of their own.

One can easily imagine that in responding to such varied needs, Horus could become very complicated. Although we managed to control complexity, we did find that the types of transformations we wanted to do on layered protocol stacks exceeded the capabilities of the available C compiler, and hence that quite a bit of hand-coding was needed to obtain high performance and to maximize flexibility.

It was in response to these considerations that Ensemble was developed, starting in the Spring of 1996. As a system, Ensemble is rather similar to Horus, although rewritten using high level programming languages and tools [Hay97]. Our insight was that much of the complexity of Horus came from overcoming inefficiencies associated with stackable protocol layers coded in the C programming language. In contrast, Ensemble's protocol suite is implemented using the O'Caml variant of the ML programming language. This language is mathematical in appearance and there are powerful theorem proving tools, notably a system called NuPrl, available for expressing transformations and other types of operations on programs coded using O'Caml. In our case, we were successful in using O'Caml to code a basic set of protocol layers for Ensemble and then using NuPrl to produce optimized and transformed versions that, in Horus, would have required hand coding and hand optimization. The NuPrl approach is automated and provably correct while the manual Horus approach was a source of bugs. Moreover, we discovered that NuPrl can potentially do quite a bit more for us.

The remainder of this paper focuses on the successes and limitations of Horus and Ensemble. Both projects are largely at an end now - Ensemble and Horus are both used by modest communities and a number of technology transition efforts should lead to their emergence in products for the mass market within the next few years. Meanwhile, our own effort at Cornell now focuses on what might be seen as third-generation issues that work to move beyond the limitations of the entire process group approach.

For example, we are increasingly convinced that virtual synchrony has some basic scalability limitations that emerge from the model itself, and our new Spinglass project[2] was born out of an insight into a new way to develop a scalable reliability model to overcome these limits. Virtual synchrony, we now believe, is simply better suited to "close grained" cooperation on a scale of tens of members (certainly, less than one hundred group members), while the protocols we are using in Spinglass provide high reliability and steady data delivery to potentially thousands or millions of recipients. We imagine Spinglass as a technology one might use side by side with Ensemble or Horus, because it offers reliability guarantees that are provably weaker than those of the virtual synchrony model, and the virtual synchrony model remains necessary in many situations. An illustration of this arises in our discussion of the Ensemble security work, which combines Ensemble groups with Spinglass protocols.

Similarly, we are continuing to work with NuPrl as a program verification and automated protocol transformation tool of unique power and flexibility. Whereas our initial work focused on using NuPrl to automate some of the protocol stack transformations needed to achieve high performance in stackable architectures, we are now pursuing a more ambitious goal: proving the correctness of the virtual synchrony implementation used in Ensemble, and perhaps of the security key management architecture running over this implementation. But this work, in turn, has revealed yet a third possible goal: the automated generation of provably-correct protocol stacks from relatively high level descriptions of goals. It thus may be possible to see Ensemble much as a compiler used to bootstrap a compilation process - one builds a basic compiler in a first language for a new language, but then implements a second compiler directly in the new language and discards the original one. Our work could yield, within a few years, a completely new and self-supporting infrastructure for building correct group-communication protocols and for optimizing them to achieve extremely high performance.

The remainder of this paper focuses on the accomplishments and limitations of Horus and Ensemble and the major technical challenges we face in transitioning the technology into major commercial product platforms, such as CORBA and COM/OLE.

III. HORUS EFFORT

Our work on the Horus system can be understood in terms of several distinct threads of activity, which were all conducted within the same framework and to a large degree interoperate: Layering and its consequences, real-time issues, security mechanisms, and work on protocol performance and scaling. We consider these in turn.

A. Layered Protocol Architectures in Horus, Performance Issues

The initial focus of our work on Horus concerned its use of layering to simplify the design of the virtual synchrony protocols. When this work was begun, we looked closely at the x-Kernel architecture [PHO89], developed at the University of Arizona by Larry Peterson with similar goals. We found, however, that the x-Kernel was designed with point-to-point TCP-style protocols in mind. For our work on group communications, a more flexible and more standardized interface to each layer was needed.

Accordingly, we developed what we now call the Horus Common Protocol Interface, or HCPI, as a standard interface to and between protocol layers. The interface provides "up", "down" and "control" API's, and operates under a model in which messages and other events travel from the user down the stack to the I/O interface, or from the I/O interface up to the user. For example, an encryption layer might receive outgoing messages from higher layers, use a key to encrypt the body of those messages, and then pass the message to the outgoing message interface of the next layer below. Incoming messages would, similarly, be decoded on arrival and discarded if corruption or tampering was detected.

Over the 7 year period since this interface was first proposed, other groups including the OMG fault-tolerance standards group and the IETF reliable multicast research task-force have proposed creating standard architectural slots similar to the ones occupied by Horus protocol layers. Our effort has offered an updated HCPI interface to these organizations, but until the present, it seems that the aggressive use of layering adopted in Horus remains more advanced that what these organizations might consider.

Layering gives rise to several forms of overhead. A message, traveling down the stack, may looked at by a whole series of layers, most of which do nothing at all to the message. When a layer does add a header to a message, it may need to assume that it is the only layer in the stack, hence to add even a single bit, a layer may need to create a header large enough to hold an integer. By the time a message reaches the wire, it may have many bytes of largely empty headers on it, and may have skipped through as many as 20 or 30 layers that basically took no action.

During 1995, one of us tackled this issue, and developed a methodology for optimizing layers to avoid both forms of overhead [vR96]. Although this work was done separately from Horus, we considered it to be part of the overall technology base. In essence, the approach involves compressing the headers by eliminating wasted space, and also separating headers into different types of data. Header information that remains constant after a stack is established is only transmitted once, and a typical message only carries headers that actually contain changing values - potentially a very small amount of data. Messages are aggregated (packed) to make optimal use of network

[2] See http://www.cs.cornell.edu/Info/Projects/spinglass/ for information and references to Spinglass publications.

packets. And, through a decomposition of each layer into data touching and non data-touching parts, it proves possible to short-circuit the path a message takes through the stack, reducing the critical path between the application and the wire to just a few instructions even for a very complex protocol stack.

With these optimizations in place, the Horus Protocol Accelerator set a number of performance records. Running over a zero-copy communications architecture called U-Net (similar to the VIA standard), this version of Horus introduced only a few microseconds of overhead beyond the overhead of the network adaptor and associated software.

B. Real-Time Cluster Computing

Unfortunately, the ability to demonstrate high performance is not enough to achieve real-time responsiveness in some critical applications. Earlier, we noted that our work on Isis was adopted by the Navy for use in its AEGIS architecture. This system includes a number of cluster-style computer systems that are used to compute tracks for airborne objects detected by the AEGIS radar, and serve as the basis of weapons targeting applications. Since threats may be moving at very high speed, real-time response is vital even when failures occur within the cluster. A similar need arises in telecommunications switching architectures, where a co-processor may be asked how a call-establishment request should be routed; the SS7 architecture used in such settings requires 100ms response times even while a failure is handled.

Working with our group, Roy Friedman explored the use of Horus as a technology for cluster control in real-time applications of this sort [FB96]. He considered two styles of solution. In the first, Horus was used to implement small process-groups of two or three processes each, using load-balancing and fault-tolerant RPC mechanisms within these to guarantee that each request would be handled even if one or more failures occurred while the cluster was heavily loaded. With this approach, Friedman was only able to achieve a throughput of a few hundred requests per second, and the Horus failure detection timer (six seconds) emerged as a performance limit: a request might potentially be delayed, if a failure occurred under heavy load, until the detection timer was triggered. For the sorts of applications just mentioned, such delays are totally unacceptable.

Friedman then developed a different solution in which Horus operates as a side-band mechanism for cluster control and data replication, but "offline" from the basic request loop. In this approach, Friedman was able to aggregate batches of requests and used hand-coded, highly optimized protocols for the basic request dispatch and handling communication paths. During the period before Horus discovered a failure, data might pile up, but Friedman used a number of compression schemes to minimize the amount and avoid overloading available buffering.

The approach was a dramatic success: for the SS7 telephone architecture, Friedman now achieved 20,000 requests per second on a 64-node cluster, demonstrated that performance improvements were possible when the cluster size was increased, and was able to sustain 100ms response times even as nodes were taken offline, crashed, or restarted while the switch was under load [FB96].

Friedman's work illustrates, for us, both the power of Horus and a limitation. The benefit of this work was that for the first time, a way to use a cluster of computers in a time-critical fault-tolerance application was demonstrated. Yet the work was technically complex and suggests that unless these methods can be embedded into a very low level of the operating system (for example, into the clustering technology of the NT Clusters system), application developers will have great difficulty exploiting the approach. Horus, viewed from the perspective of this type of real-time application, is a necessary tool, but not sufficient. On the other hand, for high-value applications such as the AEGIS tracking service, it does seem clear that Friedman's work points to a methodology for achieving very high degrees of scalability and real-time responsiveness while tolerating faults.

C. Security in Group Communication Systems

The Horus system was also the setting for our initial foray into security for groups of participants in large networks. Working with Mike Reiter [RBvR94, RB94], we developed a means of securing the virtual synchrony model itself, so that only trusted processes would be allowed to join a process group, and so that group members could obtain a shared group key. This problem involves authentication at the time of the group join, and rekeying when a member joins or leaves (so that prior communication in the group, or subsequent communication, would not be accessible to the new member).

Our work on security can be seen as complementary to work on group security arising directly from the Internet community. Recall that the DNS and routing services of the Internet replicate various forms of data. During the early 1990's it became important to secure the protocols used to update these, and the resulting key distribution and management problem became a classical topic for the security research community. Here, the notion of group membership is much weaker than the one used in the virtual synchrony community, and there is no formal semantics for the execution model. Yet the superficial aspects of the security problem are very similar: we have a group of members, we wish to authenticate joining and leaving, and we plan to use the security keys to encrypt communication within the group.

The Horus security mechanisms have advantages and disadvantages when compared to this more network-oriented form of security. The strong semantics of virtual synchrony groups certainly offers security benefits: within this model, one actually can formalize the question of which processes legitimately belong to a group and which ones do not, and when a process does belong to a group, there are strong guarantees about the state of the data it manages. But there are also disadvantages to the model, notably that it scales poorly beyond about 100 processes. Most experience with Isis was limited to groups of five to ten processes at a time [Bir99]

and it was only with great care that Isis applications spanning more than about 250 processes were developed successfully. In the Internet, 10,000 members of a DNS service might not be at all unreasonable and one can imagine services containing millions of members. Yet scaling virtual synchrony to this degree seems not to be practical. (Our new project, Spinglass, might well provide this degree of scalability, but it uses a somewhat different execution model).

IV. Ensemble Effort

Earlier, we cited the "eggs in one basket" concern in regard to distributed systems models such as virtual synchrony, or process group security. While such approaches are beneficial to the application designer, whose task is greatly simplified by the strong guarantees of the system, if the model itself is violated as a consequence of a coding error or some unanticipated bug in the protocol itself, the application's correctness or security might be compromised. One can reduce such concerns by exhaustive testing, simulation, or by writing papers in which the protocols employed by the system are presented rigorously and a formal proof of correctness is offered. Yet none of these options yields more than a modest degree of confidence in the ultimate correctness of the running code itself. Even now, more than a decade after the development of Isis, Isis applications that seek to provide continuous availability still exist, and occasionally, one of them reports a bug never before encountered. Such bugs can easily cause the entire distributed application to crash.

As noted earlier, Horus was developed as a partial response to this concern: the technology sought to simplify the monolithic structure of systems like Isis by showing how complex protocols could be broken into simple microprotocols and stacked to match the needs of an environment. Yet Horus offers little to increase the confidence of a skeptic in the ultimate correctness of the protocols and of their implementations.

Ensemble was developed primarily as a response to these concerns. Our fundamental idea was to begin using a new and extremely powerful generation of mathematically rigorous programming and verification tools as a means of moving beyond the hand-coded optimization schemes employed when performing inter-layer optimizations in Horus, and of actually proving the correctness of key components of the system. The technology evolved in several new directions, however, as time passed: we used Ensemble as the basis of initial work on a new protocol suite, and pursued a number of topics involving dynamic adaptation using Ensemble as the base. We also developed a new security architecture within Ensemble, moving well beyond the initial Horus version. This section summarizes each of these threads of research.

A. *Formal Transformation of Protocol Stacks*

Code transformation of the Horus system was impractical in part because of the choice of programming language: by coding Horus in C, we were able to achieve extremely high performance, but this language has limited capabilities for type checking and other types of correctness checking, and such weak mathematical semantics that formally expressed code transformations are largely impossible. Accordingly, a primary reason for building a new system - Ensemble - was to create a version of Horus coded in the O'Caml programming language, a dialect of ML having strong semantics and consequently suitable for analysis and transformation using formal programming tools. This decision was informed by previous success in using Nuprl to reason about a large ML system [AL92] and to reason about hardware [LLHA94]. We were also encouraged by the work at CMU by Harper and Lee on the FOX project to code protocols in SML [HL94].

Our decision to implement Ensemble in O'Caml compelled us to confront an initial challenge of a different nature. The ML family of languages is not traditionally known for high performance, and while O'Caml is compiled, we were concerned that it might not be possible to achieve performance comparable to that of Horus. Yet the verification of a system incapable of the desired level of performance would have been much less satisfying, since we hoped to demonstrate that production-quality distributed software can actually be proved correct. Mark Hayden, who coded the system, undertook a detailed study of this issue, and ultimately developed a methodology for protocol development in O'Caml that overcomes the most common efficiency issues encountered by users of the language. His accomplishment, which involved taking control of garbage collection and using O'Caml's language features very carefully, was reported in [KHH98].

Given an initial version of Ensemble, Hayden, Kreitz and Hickey set out to use a formal mathematical tool called Nuprl ("new pearl") to automate the sorts of optimizations that Van Renesse did by hand in developing Horus [vR96]. They approached this by teaching Nuprl to read Ensemble layers - in effect, Nuprl understands each layer as the "proof" of some property, namely the protocol guarantee implemented by that layer. Nuprl was then able to do several kinds of protocol transformations. For example, because a protocol stack appears as a nested function call to Nuprl, it was possible to request that Nuprl perform an inline function expansion of the code.

The basis for all formal code manipulation is a formal semantics for a large subset of the O'Caml programming language in the logical language of Nuprl. Not long ago, the formalization of such a subset would have been cutting edge research worthy of separate funding and a PhD thesis, but in this case, we were able to build on advances in understanding of formal semantics and on the richness of the Nuprl type theory. Basically the core of O'Caml is a subset of the Nuprl term language, and therefore type theory almost immediately provides a semantics for O'Caml [Kre97]. The method is now called a "shallow embedding". This method has been used to provide a formal semantics for significant extensions of ML in the direction of object orientation, see the work of Crary for example [Cra98].

Nuprl can perform partial evaluation of functions, and this opened the door to a category of optimizations similar to those used by Van Renesse. The approach begins by recognizing that as messages traverse a stack, the code path used may be a very small percentage of the code in the stack as a whole. For example, the virtual synchrony stack treats membership change events very differently from multicasts. If a message is a multicast and no membership change is occurring, the message may be nearly untouched within the stack.

A protocol stack in Ensemble looks like a set of nested function calls. Suppose that *x* is some form of outgoing event, such as a message to send or a membership change request. Then, Ensemble's job is to evaluate $f_0(f_1(\ldots f_n(x)))$, where each of the f_i is the code implementing some micro-protocol within the stack (f_0 is at the bottom and f_n is at the top). Similarly, for an incoming event, Ensemble can be understood as evaluating the function $f_n(f_{n-1}(\ldots f_0(x)))$. Now, focus on the outgoing case, and suppose we call this entire nested function *f*. Imagine that we place an if statement in front of it, as follows: "*if(is_a_msg(x)) f(x) else f(x),"* where the predicate *is_a_msg* is true for messages and false for other types of events, such as group membership changes.

Viewing Nuprl as a form of optimizing compiler, the system can be asked to partially evaluate the function under the two cases: "*is_a_msg(x)"* is true, and "*is_a_msg(x)"* is false. Under the circumstances just described, the function will collapse to just a few lines of code in the former case because so many code branches would never be executed for non-messages, and would be deleted during the partial evaluation. In effect, we've produced an extremely optimized code path for the common case where we sent a multicast.

To generate the optimized code while guaranteeing its correctness Nuprl uses two levels of formal optimizations.

- On the first, or *static* level, symbolic evaluation and logical simplification techniques are applied separately to the code of each micro-protocol. They result in formally proven *layer optimization theorems*, which show that the effect of passing an event x through the respective protocol layer, while assuming that a *common case predicate* (CCP) like "is_a_msg(x)" (or some other property of common events) holds, can be expressed by two or three lines of code. These optimizations are executed independently from the application protocol stacks and need to be redone only when the code of a micro-protocol is modified or when new micro-protocols are added to the Ensemble toolkit.

- The second, or *dynamic* level, depends on the particular protocol stacks designed by the application developers. Given the names of the individual micro-protocols occurring in this stack, it composes the corresponding layer optimization theorems into a formal *stack optimization theorem* that describes the effect of passing an event x through the whole stack while assuming that *all* individual CCPs hold. This is not trivial, because Ensemble's composition mechanism allows that events may bounce between layers before leaving the stack instead of passing straight through it. Therefore the technique is based on composition theorems, which abstractly describe the effects of composing common combinations of optimized micro-protocols.

A stack optimization theorem not only describes a fast-path through a protocol stack but also the headers that the stack adds to a message. Since typical messages should only carry headers that actually contain changing values, all constant headers are eliminated before sending the message. For this purpose, the protocol stack is wrapped with code for compressing and expanding headers and then optimized again.

In a final step the stack optimization theorems are converted into O'Caml code, which uses the CCPs as conditionals that select the bypass path in the common case and otherwise the normal stack, as illustrated in Figure 3 (The Transport module below the stack provides marshaling of messages). This program is proven to be equivalent to the original protocol in all cases, but generally more efficient in the common case.

Using this methodology [Kre99], Kreitz, Hayden, Hickey, van Renesse and graduate student Xioaming Liu showed that Nuprl can automatically achieve protocol speedups comparable to the ones that Van Renesse achieved by hand in Horus. Moreover, Liu and Van Renesse developed an version of the method for use in adapting Ensemble to match the protocol stack to the environment. With their work, one can dynamically pick a protocol stack with just the properties needed for a given setting, obtain an optimized version of the stack from Nuprl, compile the resulting O'Caml byte code into machine code, ship this code to a version of Ensemble, and run it as the protocol stack supporting some group. At present, we do the optimization offline, but the methodology could be extended to work on the fly. This work is reported in [LKvR+99].

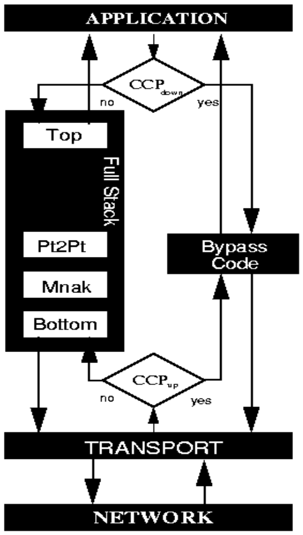

Figure 3: Optimization in the Ensemble Architecture

B. *Verification of Ensemble stacks*

Formal verification of a system such as Ensemble presents a major challenge because its many micro-protocols (currently over fifty) can be combined in literally thousands of different ways to provide a great variety of services. To verify Ensemble would mean to be able to treat the reasonable combinations of stacks and their services. Moreover, many of the individual protocols are quite sophisticated.

Additionally it is a major challenge to verify the actual system code -- indeed, most verification efforts operate by verifying the correctness of abstracted descriptions of protocol or system components. While verifying these abstractions is an interesting first step, the goal that appealed to us was to work directly with production-quality protocol implementations, and ultimately to prove that the code actually running in the system has the properties required by the user. This goal was a natural extension of our success in transforming the actual O'Caml code of Ensemble stacks.

A key step in verification is the specification of properties. This has been a lively topic in the Horus and Ensemble research. Various temporal logics were tried, including TLA [Kar97], and an axiomatic approach was considered [CHTCB96]. In the end, the ground work for our approach was laid by Hickey and Van Renesse in collaboration with Nancy Lynch, whose research effort at MIT has developed a mathematical programming language called I/O Automata, or IOA. Jointly with Lynch, they found a way to express virtual synchrony as an IOA, adapted the IOA language itself so that NuPrl could understand such a specification, and extended IOA so that it could be used in compositional settings, such as the Ensemble protocol stacks [HLV99].

We have found a particularly elegant way to formalize IOA using Nuprl class theory [BH98]. We can formalize both services and their implementations in the same style. Moreover the inheritance mechanisms of the formal class theory make it possible to inherit proofs of safety properties of a stack when new layers are added to it. This leads to a methodology that will make it easier to prove properties of stacks from proofs of components. The method seems to scale to stacks composed from many layers from a large base of micro-protocols.

When this work has been completed, we will be able to produce highly optimized executable code from provably correct protocol stacks, dynamically adapting the stack to match the environment where the protocol will be used, and providing guarantees of reliability and security formally verified by a mathematical tool and characterized in a high level notation suitable for use in reasoning about applications built over the resulting process group.

C. *Synthesis of layers*

Once we know how to prove the correspondence of code fragments to IOA statements, we can also understand this as a means for compiling from IOA into protocols (which we can optimize using our trace driven optimization methodology). So the potential exists to avoid using hand-coded Ensemble protocol stacks in favor of these more automated protocol stacks produced using NuPrl as a compiler.

The basis of this synthesis capability is the fact that global services, abstract protocol specifications and code layers are treated as modules in a common formal language. We can use our formal framework not only for verification purposes but also for the synthesis of protocols from specifications. Because the generated code is correct by construction, our framework supports the development of new communication protocols, which is a notoriously difficult task.

A synthesis of protocol layers that implement a global service will be supported by a small collection of generic methods for transforming specifications of distributed systems. This methodology is not entirely new. Synthesizing algorithms from specifications by applying specification transformations is a well-known principle in program synthesis, and synthesis from proofs has been explored for many years as well [Wal69, BC85,Kre98].The novelty of our approach lies in providing methods for the synthesis of *distributed* systems, which is by far more difficult than synthesizing serial algorithms.

We describe four of the most common generic methods.

1) Replication of Global Values

Applying this method will allow us to represent values locally. It creates a local copy of each value, ensures consistency by making each process broadcast these values, and introduces a unique token. Virtual synchrony (EVS) is a prerequisite for this step as otherwise consistency has to be ensured by more complex means.

2) Adding Fault Tolerance

This method, which requires local values as prerequisite, makes the local values fault tolerant. It forbids access to values during a view change and implements merging of multiple copies after a view change.

3) Conversion to Message Passing

This method assumes fault tolerance and generates the typical event-handler interface of Ensemble by converting each operation to a message.

4) Code Generation

is always the last step of a synthesis. It takes a specification that provides the event-handler interface and converts the abstract specification into executable code. Because of the close relation between abstract specifications and the representation of Ensemble's code in our LPE this step is straightforward.

As an example, we describe the synthesis of a total order layer from the specification of the total order *service*. The total order service has a simple specification. In addition to the properties of EVS, it also ensures that all processes in a view receive messages in the same order.

The total order specification represents this requirement with a global queue that orders all the messages sent in the view. A message can be received only if it is the next message in the global queue.

The ETO specification is not directly implementable because its global queue contains global information, so the first step is to *replicate the global value* so that each process contains a local copy of the queue. The *Replicate* generic method introduces a token to ensure atomic access and consistency of the local copies, but the copies are not robust to failures. The next step is to apply the *Fault Tolerance* generic method, which prohibits access to the queue during a view change, and creates a fresh queue once the view change is completed. At this step, we have partitioned the service into local protocol layers, and in the final step we apply the *Convert to Message* Passing generic method to convert the layer actions (which refer to semantic events like view changes) to explicit message passing style. This final step can be completely automated.

D. Scalable Security Architectures and DVPN's in Ensemble

Ensemble has also been used as a base for an expanded effort to provide security for process group applications and systems [RBD99, RBH98]. As noted earlier, our work on the Horus system resulted in an initial mechanism for securing process group communication and the group abstraction.

In our work on Ensemble, we have focused on expanding the capabilities of this basic idea and using group security keys in innovative ways. With respect to the basic security architecture, Ensemble implements a scheme (developed primarily by Ohad Rodeh, a research in Dolev's Transis group at the Hebrew University in Jerusalem) whereby asymmetric public keys can be "traded" for symmetric point-to-point keys and group keys. These symmetric keys provide a high speed path for signing messages and encrypting their contents, and can also be used by secured applications for application-specific security purposes.

Rodeh's solutions are innovative in two ways. First, he employs a hierarchy of protocols for key management and key refresh, and has a particularly fast solution to the key refresh problem when processes join or leave a group. His algorithm rekeys within milliseconds, permitting a group to (potentially) change keys so rapidly that even if a key were broken, the adversary would gain access to just one or two multicasts before a new key was substituted for the compromised one [RBH98]. Additionally, Rodeh developed a fault-tolerant extension of the well-known Wong-Gouda-Lam tree-based key management architecture, a protocol that in its original form was centralized and not fault-tolerant. Rodeh's version scales easily within Ensemble's process groups, although these remain limited to perhaps one hundred or two hundred members [RBD99].

The scalability limits of Ensemble prevent Rodeh from using this technique to secure extremely large groups, which might have tens of thousands of members. For this purpose, however, Rodeh is exploring a very simple combination of Ensemble with our new Spinglass system. As mentioned earlier, Spinglass introduces a new and extremely scalable data point in the reliable group communications spectrum, offering a suite of probabilistic protocols that can be used to communicate reliably with huge numbers of processes even in networks subject to the most extreme forms of disruption. Rodeh is linking the secured Ensemble group mechanisms to the Spinglass mechanisms so that Ensemble can control a core group and this, in turn, can manage security keys on behalf of a very large group of leaf nodes. Inspired by the DNS architecture for the Internet, this approach allows typical users to talk to a local representative of the security hierarchy and low cost, while drawing on the strong properties of Rodeh's Ensemble solution to guarantee rapid key refresh and other aspects of a strong security model.

E. Dynamic Adaptation

The third major research topic that has been explored primarily within the context of our work on Ensemble is concerned with dynamic adaptation. As described previously, the adaptation problem arises when an application expresses requirements for a process group that can be satisfied in more than one way, depending upon the environment within which the group runs [adaptive paper].

Two examples will illustrate this idea. The first is concerned with multicast ordering protocols. Over the past two decades, a great number of ordered, reliable multicast protocols have been developed. Abstractly, it is common to view such protocols as representing optimizations of total ordering. For example, in a process group where we happen to know that only one member will initiate multicasts, a protocol providing fifo ordering would actually be "strong enough" to satisfy a total ordering requirement.

Even in a group where there are multiple senders, one faces such a choice. If the senders are willing to use a token passing or locking scheme, the most appropriate total ordering protocol would be one that exploits the mutual exclusion property - these include causal ordering protocols and token-based total ordering protocols. Lacking a locking scheme, one might still use a token based protocol, but other protocols such as time-stamped ordering protocols now have potential advantages. Broadly, the best choice depends upon the nature of the application, and the nature of the environment within which we run it.

Similarly, the most appropriate security mechanism depends on the setting. Behind a firewall, one may face only a very benign security requirement, and limit "security" to some form of join authentication. Yet when the same application includes a group member running outside the firewall, it may be important to encrypt all group communication, and to authenticate even the messages used within the join protocol itself.

Virtual synchrony offers an appealing way to think about adaptation. Each time the membership of a group changes, the virtual synchrony model announces this through a new process group view, synchronized with respect to ongoing communication among group members. Normally, the view is just a list of group members, ranked in some canonical manner. Our work on adaptation extends the same idea: each time the view changes the new view also includes a new protocol stack for each of the members.

Within Ensemble, we've used this mechanism to change protocol stacks on the fly, with roughly the same (very small) overhead incurrent when group membership changes because a process joins, leaves or crashes. What we do is to associate with each stack a small software module responsible for monitoring the environment, watching for conditions under which some other stack would be more desirable than the one currently in use. When conditions change, this module triggers the new-view algorithm, arranging that the new stack be installed at all members in a virtually synchronous manner.

For example, if we are using a total ordering protocol that is appropriate only with a single sender within a group, adaptation might be triggered when a second process first attempts to send: the attempt would cause the group to switch to a new total ordering protocol, at which point the interrupted send request would be allowed to complete. If we are using a security mechanism appropriate only within a firewall, adaptation might occur when a member from outside the firewall joins the group, and so forth.

V. TECHNICAL CHALLENGES ASSOCIATED WITH TRANSITION

One of the most difficult problems we've encountered in our research has been the challenge of technology transition. Broadly, it has been our belief that unless these technologies make the transition from academic demonstrations into broad commercial availability, they will not have the degree of impact that we seek and that DARPA hopes for in projects such as this. Yet in the area of networking and distributed computing, there is tremendous resistance to doing anything other than what the basic Internet supports.

As noted earlier, our first project, the Isis Toolkit, achieved some degree of commercial uptake and had some notable successes. Through this process, Isis became a viable option for military and government projects and had an important influence on technology planning within the services, reflected in the DD-21 architecture.

However, starting a company to commercialize Horus and Ensemble was unappealing to us. Our effort has made both systems available to "all comers" with very few restrictions of any kind, and at no fee. Cornell and the original Ensemble developers - Hayden and Vaysburd - provide some support, also for free. This has resulted in some uptake of the technology, and we have a small user community that includes some large companies (notably, Nortel and BBN) and many small ones. An exciting recent opportunity involves technology transfer into the restructured electric power grid. We are pursuing this topic jointly with a consortium organized by the Electric Power Research Institute, EPRI.

To see a broad-based transition occur, companies of the size of Microsoft and Sun need to become interested in this technology area. There is some good news on this front: the industry as a whole is looking at group communication tools closely, and several standards organizations have been exploring possible standards for reliable multicast and object replication. As these trends advance, one can anticipate a first-class role for reliable multicast in standard operating systems, such as Solaris, Linux and NT, and with that development, technologies such as ours would find a natural home.

Overall, we are encouraged by these developments, but we also see an argument for winding down the effort on Horus and Ensemble in favor of new directions. Accordingly, while continuing work on the Nuprl verification methodology, our overall project has shifted attention to a new technology based on a suite of highly scalable 'gossip'_based protocols with probabilistic properties. The protocols and their behavior take us into a domain rather different from virtual synchrony, while we wait and watch to see what the ultimate impact of our work over the past few years will be.

VI. CONCLUSIONS

Cornell research has placed process group computing, especially with virtual synchrony, or a firm footing. Over the course the three projects we've conducted in this area, we've contributed techniques for achieving high performance, security, real-time guarantees and provable correctness. Although we were not able to scale virtual synchrony to very large settings, the gossip-based protocols developed in our work on Ensemble have taken on a life of their own, and form the basis of a new project (Spinglass) that promises to contribute a new and extremely scalable data point for the spectrum of reliable multicast protocols and applications.

Over the years, the Cornell effort has helped enable some very high profile applications, and we've played an instrumental role in showing that these solutions really work. Down the road we expect that the technologies we pioneered will have a good likelihood of becoming standard in products from major vendors, that IETF standards will emerge for the area, and that CORBA will provide solutions - all traceable to our work and that of related projects.

Our vision of the future is one in which reliable process group computing will reside side-by-side with scalable probabilistic technologies, of the sort now under development in our Spinglass system. Through this approach, we believe that reliable, secure distributed computing can become a commonplace reality on a wide range of systems spanning both large-scale WAN applications based on conventional workstations down to massive networks of very small devices such as might arise in future miniaturized control settings or future sensor networks.

VII. REFERENCES

[AL92] Aagaard, Mark and Miriam Leeser. Verifying a Logic Synthesis Tool in Nuprl. In *Proceedings of Workshop on Computer-Aided Verification.*, 72-83, Springer-Verlag, (1992)

[BC85] Bates, J. L. and Constable, Robert L. Proofs as Programs. In *ACM Transactions on Programming Languages and Systems* 7(1), 53-71 (1985)

[BH98] Bickford, Mark and Hickey, Jason. *An Object-Oriented Approach to Verifying Group Communication Systems* (submitted to CAV), (1998)

[Bir97] Birman, Kenneth P. *Building Secure and Reliable Network Applications.* Manning Publishing Company and Prentice Hall, (1997)

[Bir99] Birman, Kenneth P. A Review of Experiences with Reliable Multicast. *Software Practice and Experience* 29(9), 741-774 (1999)

[BJ87] Birman, Kenneth P. and Joseph, Thomas A. Exploiting Virtual Synchrony in Distributed Systems. *11th ACM Symp. on Operating SystemsPrinciples*, (1987)

[CHTCB96] Chandra, Tushar D; Hadzilacod, Vassos; Toueg, Sam and Charron-Bost, Bernadette. On the impossibility of group membership. In *15th ACM Symposium on Principles of Distributed computing*, 322-330, (1996)

[Cra98] Crary, Karl. *Type-theoretic Methodology for Practical Programming Languages*. PhD thesis, Cornell University. Department of Computer Science, (1998)

[CZ85] Cheriton, D. and Zwaenepoel, W. Distributed Process Groups in the V Kernel. *ACM TOCS* 3(2), 77-107 (1985)

[FB96] Roy Friedman and Ken Birman. Using Group Communication Technology to Implement a Reliable and Scalable Distributed IN Coprocessor. Proceedings Telecommunications Information Network Architecture 96, Heidelberg. VDE-Verlag , 25-42, (1996)

[GL99] Goft, G. and Lotem, Y. E. The AS/400 Cluster Engine: A case study. International Workshop on Group Communication (IWGC'99), (1999)

[Gla98] Glade, Bradford. *A Scalable Architecture for Publish-Subscribe Communication in Distributed Systems.* Ph.D. dissertation, Cornell University Dept. of Computer Science, (1998)

[Hay97] Mark G. Hayden. *The Ensemble System.* Ph.D. dissertation, Cornell University Dept. of Computer Science. (1997)

[Hayden-JFP] Hayden, Mark. Distributed Programming in ML. To appear: Journal of Functional Programming, 1999.

[HL94] Harper, Robert and Lee, Peter. Advanced languages for system software: The Fox project in 1994. School of Computer Science Technical Report, CMU-CS-94-104, Carnegie Mellon University, (1994)

[HLv99] Hickey, Jason; Lynch , Nancy and Van Renesse, Robbert. Specifications and Proofs for Ensemble Layers. In *5th International Conference on Tools and Algorithms for* *the Construction and Analysis of Systems*. LNCS 1579, 119-133, Springer (1999)

[Kar97] Karr, David, A. *Specification, Composition, and Automated Verification of Layered Communications and Protocols*. PhD thesis, Cornell University. Department of computer Science, (1997)

[KHH98] Kreitz, Christoph; Hayden, Mark and Hickey, Jason. A Proof Environment for the Development of Group Communications Systems. In *15th International Conference on Automated Deduction* LNAI 1421, 317-322 Springer, (1998)

[Kre97] Kreitz, Christoph. *Formal Reasoning about Communication Systems I: Embedding ML into Type Theory*. Technical Report TR 97 - 1637 Department of Computer Science, Cornell University, (1997)

[Kre98] Kreitz,Christoph. Program Synthesis. Chapter III.2.5 in *Automated Deduction - A Basis for Applications.* 105-134. Kluwer, (1998)

[Kre99] Kreitz, Christoph. Automated Fast-Track Reconfiguration of Group Communication Systems. In *5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. LNCS 1579, 104-118. Springer, (1999)

[LKvR+99] Liu, Xiaoming; Kreitz, Christoph; Van Renesse, Robbert; Hickey, Jason; Hayden, Mark; Birman, Kenneth and Constable, Robert, L. Building reliable, high-performance communication systems from components. In *17th ACM Symposium on Operating Systems Principles*, (1999)

[LLHA94] O'Leary, John; Leeser, Miriam; Hickey, Jason and Aagaard, Mark. Non-Restoring Integer Square Root: A Case Study in Design by Principled Optimization. In *International Conference on Theorem Proving & Circuit Design.* (1994)

[PHO89] Peterson, L., Hutchinson, N., O'Malley, S. and Abbott, M. RPC in the *x*-Kernel: Evaluating New Design Techniques. *Proc. 12th ACM Symposium on Operating Systems Principles,* (1989)

[PS97] Piantoni, R. and Stancescu, C. Implementing the Swiss Exchange Trading System. *FTCS 27,* Seattle, Washington, 309-313, (1997)

[RB94] Reiter, M. and Birman, K. How to Securely Replicate Services. *ACM Trans. on Programming Languages and Systems* 16(3), 986-1009, (1994)

[RBD99] Rodeh, O., Birman, K.P. and Dolev, D. Optimized Group Rekey for Group Communication Systems. To appear: 2000 Network and Distributed Systems Security, San Diego (2000)

[RBH98] Rodeh, O., Birman, K.P., Hayden, M., Xiao,. Z., Dolev, D. Ensemble Security. Department of Computer Science Technical Report 98-1703, (1998)

[RBM96] Robbert van Renesse, Kenneth P. Birman, Silvano Maffeis. Horus: A Flexible Group Communication System. *Communications of the ACM* 39(4), 76-83 (1996)

[RBvR94] Reiter, M., Birman, K. and van Renesse, R. A Security Architecture for Fault-Tolerant Systems. *ACM Trans. on Computer Systems* 12(4), 340-371 (1994)

[SWLP94] Stickel, Mark; Waldinger, Richard; Lowry, Michael; Pressburger, Thomas and Underwood, Ian. Deductive Composition of Astronomical Software from Subroutine Libraries. In *12th International Conference on Automated Deduction*, LNAI 814, 341-355, Springer, (1994)

[Vog98] Werner Vogels *et. al.* The Design and Architecture of the Microsoft Cluster Service - A Practical Approach to High Availability and Scalability. Proc 28th FTCS, Heidelberg, (1998)

[vR96] Van Renesse, Robbert. Masking the Overhead of Protocol Layering. Proceeding of the 1996 ACM SIGCOMM Conference, Stanford, (1996)

[Wal69] Waldinger, Richard, J. Constructing Programs Automatically Using Theorem Proving. PhD thesis, Computer Science Department. Carnegie-Mellon University, (1969)