# Experiences with Asynchronous Communication Models in VEOS, a Distributed Programming Facility for Uniprocessor LANs.

*Geoffrey P. Coco and Dav Lion*

Human Interface Technology Laboratory
FJ-15, University of Washington, Seattle WA 98195, U.S.A.

## Abstract

Like conventional multiprocessors, workstation clusters can provide data sharing and parallel computing. But unlike multiprocessors, these clusters provide flexible connectivity and can tolerate heterogeneous processing elements.

Uniprocessor LANs are a common choice for cost-effective computing. The workstation nodes typically run a version of Unix and support common Unix services such as reliable networking, and multiprogramming. We present VEOS, a highly portable programming facility which effectively utilizes common Unix services for distributed and coarse-grain parallel programs.

VEOS was designed for fast prototyping of distributed Virtual Environment applications across heterogeneous workstation clusters. The VEOS programming model emphasizes asynchronous communication, and distribution based on *entities*, a mechanism for non-preemptive task decomposition.

## 1 Introduction

Many academic and research sites employ Local Area Networks (LANs) of uniprocessor workstations for cost effective computing. The Human Interface Technology Lab uses LANs of commodity workstations to run Virtual Environment applications. We developed the Virtual Environment Operating Shell (VEOS) to manage distributed computation in this context.

Virtual Environment (VE) applications often utilize diverse computing resources simultaneously. For example, a single program may employ a high-performance graphics computer for binocular visual display, a medium-power workstation to track the user's position and gestures, a specific workstation as interface to dedicated voice IO devices, and a high-power general purpose workstation to compute spatial collisions and dynamics. Also, it is often desirable to include many users in the same environment and to distribute a VE program across a wider area to other sites. For these reasons, VEOS is highly portable, relying only on common Unix services.

The builders of VEs are designers, teachers, experimenters, artists, *and* programmers. Because the Human Interface Technology Lab caters to the non-technical user, VEOS attempts to provide novice programmers with simple access to the functionality though a high level (Lisp) programming interface [XLisp]. Since VE research is in a pioneering stage, many ideas need to be implemented and explored. For this reason VEOS was designed with fast prototyping in mind.

Although preemptive processes have become a dominant programming paradigm, Unix processes are known to be inefficient

for practical multiprogramming. Thread libraries require architectural specificity, so are not easily portable. Instead of processes or threads, VEOS manages collections of user-programmed non-preemptive units of distributed processing called entities. VEOS uses a single Unix process on each network node to form a pool of virtual processors upon which entities run.

An additional reason for multiplexing a single process is protection. Our network is typical of uniprocessor installations, where machines are shared among many users. VEOS and other processes on the machine are isolated and protected from each other by Unix.

In this paper, we describe the VEOS design focusing our attention on entity communication. We report on experiences programming and interacting with VEOS distributed applications. We investigate the tradeoff between potential parallelism and costly inter-node communication. We advocate asynchronous communication to maximize coarse-grain parallelism across processors, and utilization within processors.

# 2 Related Work

A distributed programming facility for the topology described above would both provide access to remote resources and support parallel computation on them. For such a system to be portable, it would need to be built on top of the existing operating systems. While modern micro-kernel architectures such as Mach [Mach] make such a design inexpensive, most existing workstation operating systems are not micro - kernels, so care must be taken to minimize costly kernel-application interactions. The widespread acceptance of user level threads is one example of this constraint, as is the trend towards user-level communications management [MIT].

Even with user-level control of communication, the cost of communication is high relative to the cost of computation in existing workstations, therefore it is essential to overlap communication with computation [Active]. The Active Messages system uses asynchronous prefetching to overlap communication with computation, but may still block if an expected message is delayed past its expected arrival time slot. Active messages are like VEOS messages, which invoke upcalls at the receiving end, however VEOS messages may incur an expensive method at the receiver, while Active Messages are intended only to provide quick access to retrieve remote data.

Scheduler Activations are another approach to overlapping computation and communication. Tight coupling between kernel and user-level allows the user-level scheduler to deschedule threads blocked on IO so that other threads may run. The threads, however, still assume a synchronous communications model [Sched].

Emerald [Emerald] supports distributed computation on a loosely coupled network of homogenous machines, Emerald uses migration of objects as the distribution mechanism, while VEOS passes messages between machines. Emerald makes extensive use of a specialized compiler and compiled run-time system, while VEOS uses an interpreted environment built on Lisp, providing a dynamic run-time environment. Like VEOS, Emerald was built on top of the existing Unix kernel, and provides multiple threads within a single heavyweight process.

Chores [Chores] provide low cost virtual processors called workers, which like entities have no stack. Workers are Presto user-level threads which yield or block; there is no time slice preemption. Entities are not subject to preemption, but may not block. Chores run on a single multiprocessor, while VEOS runs across multiple uniprocessors. Chores provides compiler support for

parallelization based on programmer hints, VEOS provides run-time support for parallelization based on explicit programmer decomposition.

Nectar [Nectar] linked existing heterogeneous workstations together to serve as a multicomputer. Nectar recognized that heterogeneous architectures provided more variety of resource specialization. Nectar used specialized auxiliary communications hardware to link machines together, while VEOS uses standard 10 MB/sec ethernet.

While newer network technologies like ATM can make LANs of workstations function as more tightly coupled parallel architecture [Nectar][Efficient], it will take time before ATM is as standard as TCP/IP is over Ethernet. VEOS favors portability more than performance, and existing bases of commodity workstations more than uncommon research boxes.

# 3 VEOS Overview

Though true parallelism is impossible on a uniprocessor, VEOS provides thread-like virtual processors, called *entities* ., which may be distributed over the set of hosts involved in an application. Entities, like Presto threads, are non-preemptive; instead they perform only short discrete tasks, yielding quickly and voluntarily. Unlike Presto, VEOS offers no support to detect a blocked entity, therefore entities use asynchronous IO to allow other entities a chance to run, as well as reduce the chance that the kernel will swap the entire node out, incurring a performance penalty.

VEOS provides a distributed execution environment for user-programmed entities. The task of writing a VEOS program is specifying the body of each entity in the system. In this way, entities are similar to objects in SmallTalk [Smalltalk].

Entities are the basic unit of process and data abstraction and were designed for flexibility. Every entity minimally consists of a unique location transparent name called an EntID. Entities use entIDs like capabilities to reference one another. Additionally, entities can have:

- *Methods*. These are like SmallTalk methods in concept. Methods are called from other entities asynchronously, asynchronously with flow-control, or synchronously, if necessary.
- *Persistent processes*. These are round-robin scheduled by VEOS and run non - preemptively on the node's single stack.

Entities receive messages through their methods. These upcalls provide direct object-oriented style communication. Methods are written in LISP, allowing them to be dynamically installed, deinstalled, or modified. Furthermore, arguments to methods can be any Lisp expression, giving the user the flexibility to pass data *or* code fragments.

At the heart of many simulation techniques is the concept of a *frame*. Roughly, a frame is a cycle of computation during which the entire simulation advances one time step. Updates are propagated at the end of a frame. VEOS embraces a flavor of frames for VE computation.

Because the nodes in a given VEOS invocation may have unequal workloads and processing speeds, each node has an independent notion of the frame rate. This allows VE builders to plan their programs so that the most critical components run on the fastest nodes (no automatic load balancing). There is a flow control mechanism for passing messages between nodes on unequal frame rates.

To make the frame concept work without preemption, VEOS entities perform discrete, atomic and repeatable tasks called *persist procs*. A node's frame rate is

determined primarily by the amount of work involved in performing all persist procs once. An example of a persist proc is:

```
(
;; poll physical device for data from dataglove
(setq  raw-data  (read-data-from-hand))

;; quickly update user's view
(send-to  renderer  raw-data)

;; parse the gesture
(setq  cooked-data  (parse-gesture raw-data))

;; dispatch command, if any
(send-to  command-engine  cooked-data)
)
```

For local entity communication, VEOS queues messages and interleaves message delivery with persist proc execution. For remote communication, VEOS uses Unix socket services to deliver messages to the proper node where message are queued as if they were local.

Although an entire VEOS application can be written in LISP, performance critical portions of entity code are rewritten in C by programmers after many experimental iterations in LISP by designers. In the billiard application described below, the collision detection routines were prototyped in Lisp, and rewritten in C to maximize performance. C functions are bound in with LISP for a consistent application programmer interface.

# 4 VEOS Primitives

In this section, we describe some fundamental system services and show their minimum execution times. In the next section, we describe how to build useful programming semantics out of these basic services.

Basic services include creating and disposing entities and sending messages. As a fair comparison to starting a null thread, we create an entity with an empty persist proc. For message passing, we use a round-trip semantic to measure performance.

Although, VEOS programs rarely use synchronous semantics, we show the data as the minimum round-trip time possible with our message passing implementation.

| | fork | join | sync msg | async msg |
|---|---|---|---|---|
| local | 10.6 | 1.4 | 2.0 | 7.2 |
| remote | 28.0 | 1.2 | 8.6 | 14.5 |

times in milliseconds

The column for *fork* show the time to create an entity locally and remotely. Times for non-trivial forks vary greatly because application specific entity initialization code is executed during the create. This could include defining methods, defuning LISP functions, initialization of devices, etc. The column for *join* shows low times because unlike the create operation, the entity dispose operation occurs asynchronously. The measured time is the time to dispatch a dispose request.

The column for *sync message* shows the round trip time for a synchronous method call. For the remote case, VEOS spin-waits for the reply. In the local case, a the function is executed immediately and the result is returned. The column for *async message* show the round trip time for asynchronous send-reply semantics. This scheme is described further in section 5.

# 5 Asynchronous Communication Models

• Client-Server

This model is based on request-reply semantics. Though similar to RPC [RPC], the sender inserts the request onto the network (asynchronous method call to server entity) but does not wait for the reply. Instead, the reply is passed by upcall to the client entity's dedicated reply method. The requesting entity must yield before the reply can be received.

The reply-catching method can have various semantics. It can restart a persist proc that was halted when the request went out, set a flag that a persist proc continually checks, or locally handle the reply in a more complete fashion, possibly generating new messages.

• Causal Ring

Messages act as tokens among the participating entities. Once an entity has received a message, it generates another message (asynchronous method call). Since method calls are guaranteed to be reliable, one initial method call (bootstrapping the token ring) will cause the token to pass perpetually.

The Causal ring also manages flow control by definition; there are only k messages among the entities where k is the number of tokens in the ring. We have found that with k>1, throughput is increased, but message convoying tends to occur which produces sporadic data flow.

• Transmit Only

This model is different from client-server and token communications because it involves one-way messaging. The transmit-only semantic is useful when sending data to output devices which are data sinks, such as graphical renderers or sound servers. Since the system interface guarantees reliable communication, the sender does not need a receipt for delivery of message. This scheme can be used for a purely data-flow application.

• Fully Connected Causal Mesh

This is a generalized causal ring. Asynchronous processes regulate their actions according to incoming messages, without an explicit token. Each incoming message drives some event, which may cause an outgoing message. Each entity in the mesh receives many messages and computes with each one. When the entity has received N updates from its neighbors, it then sends its update to N neighbors.

The Bounce sample application uses such a construct to lock-step the ball computations. Each ball considers a frame finished when it has heard from all other balls (the full set is the equivalent of a token). Within a frame, messages cause the ball to recompute its velocity based on potential collisions. When the frame is finished, the ball computes its next movement step and sends an update to each other ball.

# 6 Example Application

Bounce, our billiard ball simulation, is intended to show how VEOS facilities can be used to implement a distributed program, and where the tradeoffs between parallelism and message passing overhead lie.

Bounce runs on our LAN of heterogeneous workstations, and makes use of specialized resources on the network. A Silicon Graphics 4D/320 VGX provides graphical output, while several DEC 5000/240s provide collision and movement computation for the balls. There are fifteen

billiard balls, each one modeled by an entity. One entity provides an interface to the rendering engine, another access to a spaceball input device, and another provides a command console. All entities communicate via asynchronous messages which trigger methods on the receiving end.

The rendering and spaceball entities update as fast as possible. Balls compute a new update during each frame. Within a given frame, each ball checks for collisions upon receiving updates from other balls. Since frames are the granularity of time, all forces acting in a given frame are assumed to be coincident in time.

Note how the balls use asynchrony to maximize parallelism. Balls do not have to wait for all messages to begin acting upon them. They determine their new velocity iteratively. This process is driven by incoming updates from other balls. Once a ball determines that it has processed all messages in a frame, it computes its new position based on its already computed new velocity. The ball then sends out its updated position to the other balls and begins a new frame.

Communication between balls is an fully connected causal mesh. Communication between balls and rendering entity is a linear set of transmit-only links.

Logic for all entities is implemented in Lisp. Ball collision detection and movement code, though simple, was rewritten in C to reduce overhead.

Parallelism and message size are varied in the following four experiment combinations. In each case, the same logic and message traffic is generated. The differences are how the entities are distributed and how many message must travel between nodes.

- Sequential: all entities are on the rendering machine, sharing the VEOS process there. There is no parallelism. Communication overhead is minimal.
- Distributed: the renderer and spaceball are on the SGI, all balls are on one DEC5000. Coarse-grain parallelism, with increased inter-node communication overhead, 1 small message per ball per frame.
- Wholesale: like distributed, but an auxiliary entity runs with the balls on one DEC to bundle the single messages headed to the renderer into one large message. Inter-node communications overhead is 1 large message per frame. Slightly more processing overhead on the DEC to manage the large message.
- Parallel: renderer and spaceball on SGI, balls distributed over two DEC5000s. Increased fine-grain parallelism, increased inter-node communication as each ball now must send n/2 more messages across machine boundaries each frame.

frames per second

As expected, the Sequential condition had the worst performance. Communication overhead was minimal, but the lack of parallelism was fatal. Unexpectedly, Distributed had the best frame rate, clearly beating the Parallel condition, and just slightly above the Wholesale condition. We infer from this data that the increased number of messages in the Parallel condition negated the benefits of increased processing resources, showing that the ratio of communication to computation needs to be lower than in our sample application to achieve parallel speedups with VEOS.

Sending a single large packet seems similar in cost to several small packets. We had hoped that sending one large packet would be faster; however, we infer from this data that the network protocol overhead saved by reducing number of messages is negated by the increased overhead of large packet composition and decomposition.

# 7 Summary

In this paper we described VEOS, a system for prototyping distributed Virtual Environment applications on uniprocessor LANs.

VEOS combines object-oriented programming methodologies, non-preemptive task management, and asynchronous communications models to utilize clusters of non-dedicated heterogeneous workstations.

In the interest of portability across diverse commodity hardware, VEOS relies only on common Unix services such as sockets.

In order to utilize these workstation clusters without disrupting other users, VEOS links one Unix process per node to form a virtual multiprocessor.

In a platform-independent fashion, VEOS multiplexes the uniprocessors with entities and provides asynchronous message passing between entities. Asynchronous communication maximizes processor utilization by overlapping communication with computation.

*Entities* are flexible non-preemptive units of task decomposition designed around frame based computation, the model currently employed in simulations and VEs.

# 8 Conclusion

Our experiences show that asynchronous communication with non-preemptive task decomposition (entities) is a viable strategy when preemptive task management cannot be used.

In VEOS, inter-node communication is very expensive compared to local communication. Within one node, fine-grain task decomposition is useful and efficient as a structuring tool. Across multiple nodes, only coarse-grain decomposition is practical because the network communication overhead overwhelms the potential benefit of parallelism.

As our example application demonstrated, VEOS entities can be used to model both fine and coarse-grain task decomposition. This flexibility, together with the ease of prototyping distributed program makes VEOS a useful development environment.

# 9 References

[Active] T. von Eicken, D.E. Culler, S.C. Goldstein, K.E. Schauser; Active Messages: a Mechanism for Integrated Communication and Computation; University of California at Berkeley; ACM July 1992, pp 256-266.

[Chores] D. Eager et, J. Zahorjan; Chores: Enhanced Run-Time Support for Shared-Memory Parallel Computing; Computer Science, University of Washington; class reading.

[Efficient] C.A. Thekkath, H. Levy, E. Lazowska; Efficient Support for Loosely Coupled Multicomputing on ATM Networks; Computer Science, University of Washington; December 1992; not yet published.

[Emerald] E. Jul, H. Levy, N Hutchinson, A. Black; Fine-Grained Mobility in the

Emerald System; University of Washington; ACM Transactions on Computer Systems, Vol. 6, No. 1, Frebruary 1988, pp 109-133.

[Mach] D. Black, D. B. Golub, D. P. Julin, R. F. Rashid, R. P. Draves, R. W. Dean, A. Forin, J. Barrera, H. Tokuda, G. Malan, D. Bohman; Microkernel Operating System Architecture and Mach; Proceedings USENIX Workshop on Microkernels and Other Kernel Architectures. April, 1992; pp 11-30

[MIT] D. S. Henry, C. F. Joerg; A Tightly Coupled Processor-Network Interface; Proc. 5th International Conference on Architectural Support for Programming Languages and Operating Systems; October, 1992; pp 111-122.

[Nectar] H.T. Kung, R. Sansom, S. Schlick, P. Steenkiste, M. Arnould, F.J. Bitz, F. Christianson, E.C. Cooper, O. Menzilcioglu, D. Ombres, B. Zill; Network-Baed Multicomputers: An Emerging Parallel Architecture; Computer Science, Carnegie Mellon University; ACM July 1991, pp 664-673.

[Presto] B. Bershad, E. Lazowska, H. Levy; PRESTO: A System for Object-oriented Parallel Programming; Computer Science, University of Washington, Software-Practice and Experience, Vol. 18(8), August 1988, pp 713-732.

[RPC] A. Birrel, B. Nelson; Implementing Remote Procedure Calls; Xerox Palo Alto Research Center; ACM Transactions on Computer Systems, Vol. 2, No. 1, February 1984, pp 39-59.

[Sched] T. E. Anderson, B. N. Bershad, E. D. Lazowska, H. Levy; Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. ACM Transactions on Computer Systems 10, 1; February, 1992; pp 53-79

[Smalltalk] A. Goldberg; Smalltalk-80; Xerox Corporation; Addison Wesley, 1984;

[VPL] Virtual Reality data-flow language and runtime system, Body Electric Manual 3.0; VPL Research, Redwood City, CA; February 1991.

[XLisp] XLISP 2.1 by David Betz, User's Manual.