A DSM Cluster Architecture Supporting Aggressive Computation in Active Networks

Peter Graham[*]

Parallel and Distributed Computing Laboratory
Department of Computer Science
University of Manitoba
Winnipeg, Manitoba, Canada R3T 2N2

pgraham@cs.umanitoba.ca

Abstract

Active networks allow computations to be performed in-network at routers as messages pass through them. Active networks offer unique opportunities to optimize network-centric applications in ways that are not possible using conventional networks. Unfortunately, the need to route packets at full network speed means that very little computation can be done per packet, per router. This seriously restricts the range of in-network applications that can be developed. Computationally intensive applications are restricted to executing outside the network and thus many potential in-network optimizations are precluded.

We propose a scalable cluster architecture using software Distributed Shared Memory (DSM) that can be used as an "attached processor" at routers for executing active code. This novel application of DSM enables the construction of aggressive active network protocols by providing significant compute capacity outside the router's critical packet routing path. The use of DSM simplifies the implementation, and extends the capabilities, of the active packet execution engine in ways that a message passing cluster cannot. Further, the characteristics of active processing enable specific optimizations to consistency maintenance.

Keywords Active Networks, Distributed Shared Memory, Cluster Computing, Memory Consistency.

1. Introduction

This paper proposes a DSM-based cluster architecture for executing computationally intensive code in active networks. Active networks permit code to be injected into the network that will be executed at the routers that comprise the network. Being able to execute code to dynamically modify the behaviour of the network, and/or the data it carries, offers opportunities for optimization of many new network-centric applications.

With the exception of research test beds, current routers are passive (i.e. non-active). Further, current proposals for active router designs advocate simple extensions of passive routers where the active computation must be done by the router itself. In combination with the need to route packets at full network speed, this greatly limits the processing that may be done by active routers.

The need to perform aggressive computation in-network is due to the requirements of a wide range of new network applications that place significantly higher demands on the network than do traditional applications. An excellent example of such an application is tele-surgery which seeks to use the Internet to allow close collaboration between surgeons and remote consultants during difficult surgical procedures. A tele-surgery application requires the transmission of multiple video streams (for different camera angles) as well as audio streams and other data (e.g. patient records, clinical images, vital signs, etc.) over the network. The transmission of this data requires significant network bandwidth (a limited commodity that must be carefully managed to avoid adversely affecting other network users).

One way to address this problem of limited bandwidth is to dynamically modify the multimedia streams in-network to reflect individual user requirements [22]. If a receiver of the transmission only requires limited resolution for some video stream(s) then delivering *reduced* streams will save network bandwidth. The best place to modify the transmission is within the network at the routers that form the multicast tree [35, 5, 18, 25] that delivers the data. Unfortunately, re-quantizing video (to modify the resolution) is too

[*] This research was supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada under Grant OGP-0194227.

time-consuming to be done by a conventional router.

To perform demanding, in-network computations for tele-surgery and other applications (e.g. distance education, Internet Television, etc.) some sort of "attached" computation engine must be added to active routers to offload the computation from the router itself. We propose a Distributed Shared Memory (DSM) based Beowulf [6] style cluster interconnected using high-bandwidth, low-latency system area network (SAN) technology to implement the attached processor.

Using DSM provides distinct benefits over other alternatives (including a message passing cluster). These include:

- Simpler Implementation – Since cluster nodes share memory, state information determined at one node is easily available at others without nodes needing to be aware of one anothers' existence.
- Easier Load Balancing – Since data is equally available to all nodes the selection of a node to perform an active computation can be based purely on load.
- Improved In-Network Algorithms – DSM permits cluster nodes to share resources effectively. This can lead to improvements in active algorithms and enable new algorithms that would not otherwise be possible.
- Improved Security – Executing active code off-router means the router is not prone to failure due to faulty or malicious active code. Further, the computation power offered by a cluster permits the use of more secure but less efficient computation environments (e.g. Java).

The characteristics of active processing also create interesting opportunities for optimized consistency protocols.

1.1. Organization

The rest of this paper is organized as follows. In Section 2 related work is reviewed. Section 3 discusses the active network environment. Our proposed architecture is presented in Section 4 and we discuss the status of our prototype implementation in Section 5. We present our coclusions and discuss areas for future research in Section 6.

2. Related Work

2.1. Cluster Computing

With the development of relatively low-cost, high performance local area networks (e.g. Myrinet [9]), parallel computing with collections of workstations has become practical for many computational problems. While still lagging behind more tightly coupled parallel machines in terms of raw performance, "clusters" [11, 12] offer good performance at extremely low cost. A number of different cluster architectures have recently been proposed ranging from shared networks of workstations (e.g. NOW [4]) to dedicated cluster machines as in the Beowulf [6] approach.

Generally, cluster programming to support high performance computing has been done using message passing systems such as PVM [19], MPI [23] and HPVM [16]. While message passing systems are efficient and well suited to cluster environments, it is generally agreed that they are more difficult to program than shared memory multiprocessors. To address this problem, a number of software Distributed Shared Memory systems have been developed that run effectively on clusters.

2.2. Distributed Shared Memory

Distributed Shared Memory (DSM) provides the illusion of a shared memory programming environment where no physical shared memory exists. DSM systems can be implemented in either hardware or software. Using software DSM, the focus of this paper, the machines in a cluster maintain copies of shared data in their local memories as required. The DSM system tracks the activity of processes on each machine to ensure that all processes see only up to date copies of the shared data. This is accomplished by "sending" updated data to other machines that need it.

Over the last several years, the development of software DSM systems has been widely investigated. Following the pioneering work of Li and Hudak [26] a large body of knowledge has been created focusing, primarily, on improving the performance of such systems by reducing the number and size of messages sent. This has been accomplished using weak consistency protocols (e.g. [14, 24, 7]) as well as other optimizations (e.g. [15, 2, 3, 27, 30, 8]). Work has also been done on decreasing the overhead of the transmission protocol used to send the required consistency maintenance messages [34, 33, 10].

2.3. Active Networks

Active networks[13, 29, 32, 36] allow customized (possibly application specific) programs to execute in the network. By allowing the execution of user-defined programs, the function of the network is no longer restricted to packet delivery alone and this creates opportunities for optimizations to many distributed applications.

Code distribution techniques allow active services to be automatically and dynamically transferred to the network nodes where they are needed. With code distribution it is unnecessary to deploy all (any) services to network nodes in advance. Thus, nodes not requiring certain services do not incur the overhead of storing their code. Dynamic code

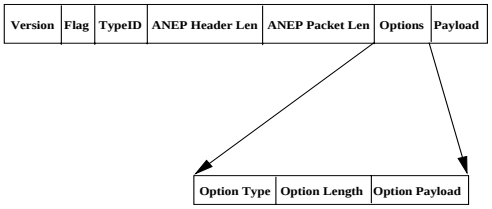

Figure 1. Format of an ANEP header

distribution also makes it is easy to develop, maintain, and upgrade *new* network services and to spread them throughout an active network. ANTS[37], a prototype active network architecture from MIT, provides on-demand, Java-based code distribution.

Active packets can be encapsulated in an IP packet using the Active Network Encapsulation Protocol (ANEP)[1] thereby making them transparent to network nodes that do not support them. This will allow the gradual deployment of active routing elements into the Internet. Non-active network nodes will simply ignore the active fields (since they are invisible to them). The `Option Type` field in an ANEP header (Figure 1) determines how active nodes should handle a given active packet. If a service corresponding to the specified type does not exist at an active node it must be automatically loaded.

Active networks are generating significant interest in the network research community. This interest is reflected in the development of anetd [17] (an active version of Unix's inetd server) as part of the ABONE active networks research testbed. Broad application of active networks, however, is currently restricted by the limited computation that can be performed at each router.

3. The Active Network Environment

To broaden the possible range of applications that can benefit from active network technology, a new active router architecture must be designed that offloads active processing from the router's critical path so that more computation can be done per packet. For security and reliability reasons, it is also important to isolate the active processing engine from the normal routing functions. To address these issues, other researchers have suggested that the active execution engine be structured as a processor "attached" to the active router (e.g. [28]). In this section we examine the requirements of such an architecture given the characteristics of the network environment in which it must operate.

Active routers (or their attached processors) must execute the "active code" specified by the `Option Type` field of each ANEP packet. This code may already be resident in the active router's memory or it may have to be loaded on demand. The processing resulting from the reception of an active packet is intended to permit dynamic modification of the *handling* of the payload data contained in each packet. This may or may not actually modify the data itself.

Depending on what is to be done, active processing may require data and code available in the packet being processed and/or it may require locally cached information (in the router's memory) and/or it may require information that is dynamically gathered in response to the packet's arrival (e.g. from other, nearby active nodes). Additionally, some of the information required may be application and/or transmission/stream specific rather than packet specific.

3.1. Active Code

Active programs that execute on active router nodes (or their attached processors), must have certain fundamental characteristics. Since the applications that will use active networks and their requirements are hard to predict, any language for specifying active code must be both expressive and composable. Some existing in-network computing systems offer only a fixed library of, typically routing-related, functions that may be invoked. Such an approach is too narrow to support the wide range of applications targeted in this paper – a powerful, general purpose language is required. To allow new applications to be supported effectively, an active programming language should also provide re-use to support the construction of new functionality using existing code. Such composability also offers the potential to dynamically combine active code for different applications that are to be applied to a single data stream.

Active code must also be portable, secure, and dynamically loadable. Portability is required because in an Internet environment, few assumptions can be made about the execution platforms that will be encountered (e.g. Cisco vs. 3com vs. Allied Telesyn routers). Thus, it must be possible to run an active programming environment easily on all potential platforms. Security is a concern due to the shared nature of routers. Unlike, workstations, routers and other network devices are potentially shared by tens of thousands of users from many different organizations[1]. The failure of active code that is processing a given packet must not interfere with the successful routing of other packets (active or passive) that happen to be passing through the same router concurrently. Finally, active code must be dynamically loadable to avoid the need to pre-distribute and store all potential active code to all possible active routers.

Historically, active code has also had to be small to meet the tight memory constraints on routers and be highly tuned for efficient execution since few router cycles could be de-

[1]Consider the case of a core router handling packets passing from a popular web site to browsers around the world.

voted to active processing. In fact, some active network systems actually deliver the active code to be executed in the `Option Payload` field (refer to Figure 1) of the packet thereby minimizing storage overhead. This approach consumes unnecessary network bandwidth but does address the conflict between lack of storage and the latency of loading non-resident active code. These requirements will, of course, be obviated to some extent when an attached processor architecture is considered.

3.2. Active Data

Unlike many more general computing environments, the execution of active code deals with several different and easily identifiable types of data. These distinct data types arise due to the characteristics of various active applications. Active network applications have been proposed that require data to be stored that is packet, application, or transmission specific. Some data is also specific to the router where the active code is executing. For example, current routing conditions (e.g. number, type, and size of packets recently processed, routing tables, etc.) is specific to each router. Such data is useful to alsgorithms such as active congestion control.

Packet specific data is usually transient and is often carried with the packet itself. An example of packet-specific data is the packet's destination address. An in-network packet-redirection application designed to send HTTP packets to alternate, more lightly loaded web servers, might modify a packet's destination IP address before forwarding the packet on to the new destination.

An example of stream specific data arises when trying to adjust video characteristics (e.g. resolution) in network. Requantization of video images is normally done using data from multiple consecutive video packets because a single packet seldom contains en entire video frame. Thus, multiple packets (and corresponding video information) must be buffered to permit re-quantization. This means *stream* specific data must be maintained.

Application specific data may be thought of as an extension of local data where an in-network algorithm (which is applicable across all active nodes) requires the storage of non-local but not stream-specific data. An example where application specific data might arise is in attempting to schedule parallel processes onto available processors in a hierarchical network [21]. In this case, each active node must maintain data about predicted future computational resources that are available "beneath" it in the network so they may be matched against requests for service. This data is application, not stream specific and is non-local since nodes must maintain information from lower nodes as well.

Another, unusual, type of data that is common in active network applications is what is referred to as "soft state" data. Because there has, historically, been limited storage available at active router nodes, many active algorithms have been designed to operate given only imprecise data. An active node's memory (its "soft store") typically contains only the most recent data even if it is incomplete. This allow the router to easily discard old data when available memory space becomes low. The reasoning behind this strategy is that older data is less likely to be important. This concept works well for active algorithms that operate on data that describe current network conditions. When router memory is relatively plentiful, the data is more comprehensive. When memory is at a premium, the data is less comprehensive but still captures the most recent conditions (and can be replenished as needed). With an attached processor architecture available memory is less of an issue but for many applications, soft data still makes sense.

3.3. Requirements Summary

The general requirements of an attached cluster architecture for active code execution can be derived from the preceding discussions of the characteristics of active code and data. First, any execution engine must be capable of executing active code expressed in a language which is general purpose and composable. This means that any execution environment must be powerful and flexible. Second, the execution environment must offer support for portability, security and dynamic loading. To meet these requirements, several active networks prototypes have used Java [20] as the language for specifying active code. Java is a powerful programming language that supports composability through its object oriented features. It also offers portability ("Write once, run anywhere"), security (type-safety, lack of pointers, etc.), and dynamic loading (e.g. the `ClassFor` mechanism). The efficient execution of programs in Java (or other, similar languages) to obtain their desirable characteristics (power, portability, security, etc.) requires significant compute and memory capacity.

Other architecture requirements can be inferred from the characteristics of the network environment in which active routers must operate. Fundamental among these is the requirement for significant compute power to permit processing of active packets at network speed[2]. Even with simple processing per packet, the volume of packets routed accounts means significant compute power is needed. When complex, active processing is required for each packet this requirement increases sharply. Further, as the frequency and complexity of in-network processing grows, significant demands will also be placed on router memory capacity. Greater capacity will be needed to store the necessary packet, stream, and application specific data as well as to cache the large collection of frequently used active routines.

[2] Core router nodes must process millions of packets per second.

One way to address these requirements is to exploit parallel execution. Wetherall [36] recognized that much packet processing can be readily parallelized since there is no tight synchronization required between individual network transmissions. This suggests that some sort of parallel architecture be used to implement the attached active processing engine for each active router.

The attached processor architecture for active routers must be relatively inexpensive, upgradeable, scalable, and easy to maintain. Further it must be easy to integrate with existing routers to avoid high cost and difficulty in deployment (the Internet is a 24x7 service). Rather than designing an active router that incorporates a new, custom-designed multiprocessor for active processing, it makes sense to use commercial off-the-shelf (COTS) components to build clusters for the execution of active code. Clusters offers excellent performance for applications that are not communication intensive and are extremely cost-effective. Further, they can be maintained easily, are easily upgraded (e.g. faster CPU, more memory, etc.) and scale well to dozens or even hundreds of processors for the sort of application required. Since most existing routers provide the ability to redirect certain packets to an external machine for processing it will be fairly simple to use a cluster as an attached processor.

To be effective as an active router's attached processor it must be possible for the active router to easily exploit all processors in the cluster effectively. With a conventional, message-passing, cluster environment this could be difficult to do. Using a DSM-based cluster offers many advantages that are described in the next section.

4. A DSM Cluster based Active Node

4.1. Benefits of a DSM-based Cluster

A DSM-based cluster provides many benefits not offered by a message-based cluster for implementing an active processing engine while retaining the normal cluster benefits (cost, scalability, etc.).

A primary benefit is that the implementation of the active engine is simpler both for the active router and the cluster nodes. The active router must assign work (active packets) to cluster nodes. With a DSM-based cluster, there is no need to differentiate between cluster nodes when doing this. This is particulary important in the handling of stream-specific data. Stream-specific data is created at the cluster node that the first packet in the stream passes through. This introduces a problem in a non-DSM cluster if subsequent packets in the same stream are assigned to other cluster nodes since the required stream-specific data will be stored in the original node. This means the active router must be stream aware so it can route all packets from a given stream to the cluster node holding data for that stream. This approach can also lead to load anomalies since forcing all packets in a long-running stream to be executed by a single cluster node may result in poor load balance across the cluster nodes.

It is also critically important for active routers to assign active packets to cluster nodes with the least possible overhead. To avoid making active routers stream-aware, the cluster nodes could be forced to explicitly transfer stream-specific data on demand. Unfortunately, this would significantly complicate the programming of stream-specific active applications. Using a DSM-based cluster, such transfers are effected by the underlying DSM software and are thus transparent to the active code.

The use of a DSM-based cluster also provides the ability to easily share memory capacity (i.e. one cluster node can effectively cache data in another node's memory). This means that a DSM cluster execution engine has greater aggregate storage capacity and this will permit much more effective caching of active code and data which will result in fewer faults and better overall performance.

Additional benefits offered by the use of a DSM-based cluster include backward compatibility with existing active code (which expects to run in a shared memory environment), improved security, and more effective resource management. Security, as described earlier, is a critical issue in an active processing engine. Using a cluster architecture means that the active code runs off-router. A DSM-based architecture further permits code which has been authenticated and verified by one cluster node to be immediately used safely by other nodes. The availability of shared memory also makes it easy to use any spare cluster cycles on improved resource management algorithms.

The most common complaint with DSM systems is poor performance relative to message-passing systems. This is not an issue in our environment for two reasons. First, the processing of individual packets is relatively independent compared to tightly coupled processes solving numeric problems (the normal environment in which DSM systems perform poorly). Second, the unusual ability to characterize specific active data types means that there are many opportunities to exploit customized consistency protocols.

4.2. The DSM Cluster Architecture

A high level diagram of our proposed DSM-based cluster architecture is shown in Figure 2. The Active Network Node (ANN) routes packets between network nodes (hosts or other routers). Packets that are active are redirected at high speed via the Cluster Switch (CS) to a Cluster Node (CN) for execution. The selection of a CN to execute a given active packet on may be made to achieve optimal load balancing since all state information that may be needed to execute the active code is equally available at all cluster nodes. Once active processing is complete, the (possibly

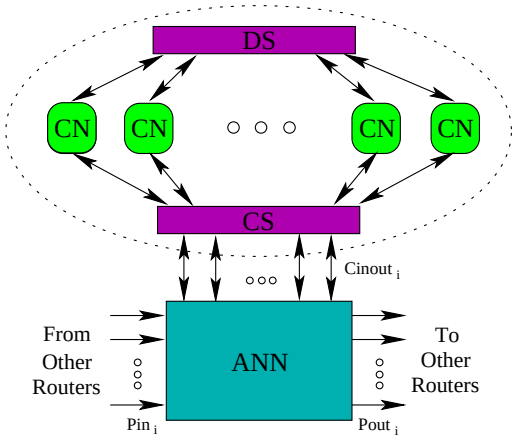

Figure 2. Active Cluster Architecture

```
loop forever do
    if (i=Msg.Avail(Pin*) then
        if (Msg.Type(Pini)=Active) then
            cn=(NxtCN=(++NxtCN)%NumCN);
            Send Msg(Pini) on Cinoutcn
        else /* must be passive */
            Route Msg(Pini) to
               appropriate Poutj
        endif
    endif
    if (i=Msg.Avail(Cinout*) then
        Route Msg(Cinouti) to
           appropriate Poutj
    endif
endloop
```

Figure 3. Active Router Processing

modified) active packet is returned to the ANN via the CS to be routed on towards its destination.

Network traffic required to maintain DSM consistency is carried on a separate network – the Data Switch (DS) in Figure 2. This ensures that changes in active packet arrivals and memory consistency traffic do not interfere with one another. More importantly, it will permit future exploration of customized protocols for each type of transmission.

Each router (ANN) must examine packets passing through it to determine whether or not they are active. Non-active packets are routed as usual bypassing the DSM cluster. Active packets are redirected to CNs in round robin fashion. This achieves reasonable load balancing (under most conditions) at full network speed and can be easily implemented in commercial routers. Further, existing routers designed to forward selected (i.e. active) packets to a *single* machine can still use this architecture by adding a machine between the ANN and CS which does the scheduling.

Pseudo-code for the active router function (which refers to components in Figure 2) is provided in Figure 3.

4.3. Cluster Consistency Management

The fact that the different types of data processed by active routers can be easily distinguished provides a unique opportunity to optimize consistency management between cluster nodes. Specific consistency protocols may be applied to each type of data and since only one protocol is used for any data type, the normal problems with using multiple consistency protocols concurrently do not arise. Thus, the best protocol for each type of data may be used.

To simplify system implementation, it must be possible to ensure that the appropriate protocol can be selected without the DSM system needing to be data type aware. This can be addressed by partitioning the virtual address space so that specific regions hold only a single type of data. The DSM system can then safely and efficiently select the appropriate consistency protocol based solely on virtual address.

A potential problem arises in partitioning the address space into regions, one per active data type. With a small address space, static partitioning may mean that the "space" available for certain types of data is unduly constrained. For example, if multimedia delivery is common, the region assigned to the needed stream-specific data may overflow while the region assigned to packet-specific data may be underutilized. In execution environments with 64 bit virtual addresses, there is sufficient address space to permit static division without overflow concerns. In 32 bit environments, however, dynamically adjustable regions are needed to optimize use of the available address space.

We now consider consistency management for each type of active data.

Active *code* is freely cacheable. Thus, it can be placed in an address range where *no* consistency maintenance is performed but where replication is permitted. Code that is required at each node will naturally tend to be cached in the RAM of that node thereby improving performance. The probability of finding a cached copy of infrequently used code is also enhanced in this scheme since if the code remains memory resident in at least one cluster node, it is available for use by all nodes without re-loading the code from a remote network host. This will also tend to improve active execution performance.

Stream specific data is characterized by the fact that only one cluster node will ever be accessing it at a time. This means that a consistency protocol for such data can be optimized to deliver updates to a *single* subsequent accessor. Unfortunately, the cluster node which will next access data for a given stream cannot be predicted, otherwise a sim-

ple forwarding scheme could be used. Instead, a consistency protocol based on, for example, Entry Consistency (EC) may be employed. In essence, a logical lock can be assigned to each active stream passing through the router. A lock corresponding to a given stream is considered to be acquired by a cluster node when an active computation on that relevant stream is scheduled on it. When the computation is complete, the corresponding lock is released. In accordance with the EC protocol, during a lock acquisition following a previous release, the corresponding data (the stream-specific data for the stream being "locked") is transferred to the node acquiring the lock. For long-lived data streams (e.g. corresponding to streaming video for teleconferencing or other such applications) it is likely that all the nodes in a small cluster may store copies of (various versions) of the data for a stream. In this case, protocol optimizations such as the use of deltas may also be exploited.

A new and interesting class of data is the so-called "soft-state" discussed earlier. Much of this data is used in algorithms that do not assume precise data accuracy. Instead, *rough* data approximates the actual state of the system. It may therefore be possible to maintain what we refer to as "loose consistency" for soft-state data. A loose consistency protocol is one where consistency is maintained immediately only when it is efficient to do so. When it is inefficient to maintain consistency, a local copy of data which is stale (perhaps within an age threshold) may be used instead of waiting for the most up-to-date version of the data.

This can be implemented using a lock-acquisition primitive that supports a timeout. If lock acquisition is successful, the latest version of the data will be delivered and used. If the lock times out, the DSM system must create an asynchronous handler to later receive the new version of the data and update the locally cached copy. The requesting process proceeds using the stale version of the soft state data. If there is no cached copy, the requesting process waits until the up to date copy arrives. Only local synchronization is required to ensure the update handler and the original process do not interfere with one another.

The use of loose consistency is particularly important for time-critical active network code that accesses large collections of shared data. The challenge in implementing loose consistency is tracking the most recent version of each shared datum at low cost. In the proposed active execution architecture we exploit the availability of the high speed cluster interconnect to support the many small messages that must be sent to track data versions.

5. Implementation Status

We have begun the construction of a "proof of concept" implementation of the proposed active cluster architecture. Our testbed platform is an 8 node, 16 processor Pentium-III cluster with Myrinet, SCI, and Giganet interconnections. We are currently using Myrinet to connect our active router, a Linux machine running anetd, to the cluster nodes (CS in Figure 2) and Giganet to support DSM across the cluster nodes (DS in Figure 2). We are in the process of porting Quarks [31] to run over Giganet so it can serve as our initial DSM implementation system. Quarks was chosen since it is a reasonably simple DSM system and its source code is in the public domain. We expect to have the port stable by early summer at which point we will begin modifying Quarks to optimize consistency maintenance for the various active router data types described earlier.

6. Conclusions and Future Work

In this paper we have presented a DSM-based cluster architecture that makes a highly efficient attached processor for active routers. This architecture meets all the requirements for active processing described in the paper by exploiting DSM. We have also described how multiple consistency protocols can be correctly applied concurrently by exploiting certain properties of active processing.

There are several areas of possible future research that we are exploring. These include:

- Use of cluster node disks to build a cluster-wide file system to cache active code. This would improve performance by reducing code reloads from remote hosts.
- Pre-fetching (or pre-loading from disk) of active code based on long-term active code usage patterns.
- Exploiting multi-processor cluster nodes (e.g. using processor affinity tools, such as `pset`, a just-in-time Java compiler can be run on one processor to compile active code so it will run more efficiently on another).
- Customized low-level network protocols for consistency maintenance. In particular, we are considering an active *message*-based [34] consistency protocol for stream-specific data.
- The pipelining of active processing across cluster nodes for complex, decomposable functions. Such pipelining would be managed by a new DSM protocol
- Making the timeout value in loose consistency a *per-object* parameter.

References

[1] D. S. Alexander, B. Braden, and C. A. Gunter. Active Network Encapsulation Protocol (ANEP). Active Networks Group, RFC Draft, July 1997.

[2] C. Amza, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. Software DSM Protocols that Adapt between Single Writer and Multiple Writer. In *Proc. 3rd Symp. on High Performance Computer Architecture*, pages 261–271, Feb. 1997.

[3] C. Amza, A. L. Cox, K. Ramajamni, and W. Zwaenepoel. Tradeoffs between False Sharing and Aggregation in Software Distributed Shared Memory. In *Proc. of the Sixth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'97)*, pages 90–99, June 1997.

[4] T. Anderson, D. Culler, and D. Patterson. A Case for NOW. *IEEE Micro*, 15(1):54–64, Jan 1995.

[5] T. Ballardie, P. Francis, and J. Crowcroft. Core based trees (cbt) an architecture for scalable inter-domain multicast routing. In *ACM SIGCOMM'93*, pages 85–95, 1993.

[6] D. J. Becker, T. Sterling, D. Savarese, J. E. Dorband, U. A. Ranawak, and C. V. Packer. Beowulf: A Parallel Workstation for Parallel Computing. In *Proceedings of the International Conference on Parallel Processing*, 1995.

[7] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *Proc. of the 38th IEEE Int'l Computer Conf. (COMPCON Spring'93)*, pages 528–537, Feb. 1993.

[8] R. Bianchini, R. Pinto, and C. L. Amorim. Data Prefetching for Software DSMs. In *Proc. of the Int'l Conf. on Supercomputing'98*, July 1998.

[9] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. J. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet - A Gigabit-per-second Local-Area Network. *IEEE Micro*, 15(1), Feb. 1995.

[10] P. Buonadonna. Implementation and Analysis of the Virtual Interface Architecture. In ACM, editor, *Proceedings of the 1998 ACM/IEEE SC98 Conference, Orlando, Florida, USA, November 7–13, 1998*, 1998.

[11] R. Buyaa, editor. *High Performance Cluster Computing (vol 1): Architectures and Systems*. Prentice Hall, 1999.

[12] R. Buyaa, editor. *High Performance Cluster Computing (vol 2): Programming and Applications*. Prentice Hall, 1999.

[13] K. Calvert, S. Bhattacharjee, E. Zegura, and J. Sterbenz. Directions in Active Networks. *IEEE Communications*, 1998.

[14] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proc. of the 13th ACM Symp. on Operating Systems Principles (SOSP-13)*, pages 152–164, Oct. 1991.

[15] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Techniques for Reducing Consistency-Related Communication in Distributed Shared Memory Systems. *ACM Trans. on Computer Systems*, 13(3):205–243, Aug. 1995.

[16] A. Chien, S. Pakin, M. Lauria, M. Buchanan, K. Hane, and L. Giannini. High Performance Virtual Machines (HPVM): Clusters with Supercomputing APIs and Performance. In *Proc. 8^{th} SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[17] S. Dawson, M. Molteni, L. Ricciulli, and S. Sui. User Guide to Anetd 1.6.3. SRI International, Sept. 2000.

[18] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C. Liu, and L. Wei. An architecture for wide-area multicast routing. In *SIGCOMM'94*, pages 126–135, 1994.

[19] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM Parallel Virtual Machine – A User's Guide and Tutorial*. MIT Press, 1994.

[20] J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java Language Specification (2ed)*. Addison-Wesley, 2000.

[21] P. Graham and R. Singh. A Mechanism for the Dynamic Construction of Clusters Using Active Networks. In *submitted to the International Symposium on Cluster Computing and the Grid*, 2001.

[22] P. Graham, X. Zhou, and R. Eskicioglu. Supporting Multimedia Applications Using Active Network Based Dynamically Adaptive Multicast Trees. In *submitted to the International Workshop on Intelligent Networking (IN2001)*, 2001.

[23] W. Gropp, E. Lusk, and R. Thakur. *MPI-2 Advanced Features of the Message-Passing Interface*. MIT Press, 1999.

[24] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, pages 13–21, May 1992.

[25] S. Kumary and P. Radoslavov. The masc/bgmp architecture for inter-domain multicast routing. In *SIGCOMM'98*, 1998.

[26] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. In *Proc. of the 5th Annual ACM Symp. on Principles of Distributed Computing (PODC'86)*, pages 229–239, Aug. 1986.

[27] P. Lu. Aurora: Scoped Behaviour for Per-Context Optimized Distributed Data Sharing. In *Proc. of the 11th Int'l Parallel Processing Symp. (IPPS'97)*, Apr. 1997.

[28] D. Raz and Y. Shavitt. An Active Network Approach to Efficient Network Management. Technical Report 99-25, DIMACS, 1999.

[29] J. Smith. Switchware: Accelerating Network Evolution. Technical Report MS-CIS-96-38, University of Pennsylvania, CIS Department, 1996.

[30] W. E. Speight and J. K. Bennett. Reducing Coherence-Related Communication in Software Distributed Shared Memory Systems. Technical Report ECE TR-98-03, Electrical and Computer Engineering, Rice University, 1998.

[31] M. Swanson, L. Stoller, and J. Carter. Making Distributed Shared Memory Simple, Yet Efficient. In *Proceedings of the 3^{rd} International Workshop on High-Level Parallel Programming Models and Support Environments*, 1998.

[32] D. L. Tennenhouse and D. J. Wetherall. Towards an Active Network Architecture. *Computer Communication Review*, 26(2), Apr. 1996.

[33] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User Level Network Interface for Parallel and Distributed Computing. In *Proc. of the 15th ACM Symp. on Operating Systems Principles (SOSP-15)*, pages 303–316, Dec. 1995.

[34] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–267, Gold Coast, Australia, May 1992.

[35] D. Waitzman, C. Partridge, and S. Deering. Distance vector multicast routing protocol. RFC 1075, Nov. 1988.

[36] D. Wetherall. Active Network Vision and Reality: Lessons from a Capsule-Based System. In *17th ACM Symposium on Operating System Principles (SOSP99)*, pages 64–79, 1999.

[37] D. Wetherall, J. Guttag, and D. Tennenhouse. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In *1st Open Architectures and Network Programming Conference (OPENARCH'98)*, pages 117–129, 1998.