D-Stampede: Distributed Programming System for Ubiquitous Computing *

Sameer Adhikari, Arnab Paul, Umakishore Ramachandran †

Abstract

That the future of information technology will be dominated by invisible or pervasive computing is a belief that is being shared by several research groups. In a sense this is the limit of the ubiquitous computing vision that was proposed by Mark Weiser in his 1991 Scientific American article on "The Computer for the 21^{st} Century". The hardware advances, falling price of computing and communication gear, and the presence of technology in several aspects of our everyday life is giving credence to this vision. We focus on an important problem in this space, namely, distributed programming support for an assemblage of interconnected heterogeneous computing elements that make up such an environment. We address the interactive, dynamic, and stream-oriented nature of this application class and develop appropriate computational abstractions in the *D-Stampede* distributed programming system. The key features of D-Stampede include indexing data streams temporally, correlating different data streams temporally, performing automatic distributed garbage collection of unnecessary stream data, supporting high performance by exploiting hardware parallelism where available, supporting platform and language heterogeneity, and dealing with application level dynamism. D-Stampede is currently in use at Georgia Tech for developing instances of ubiquitous computing applications. We discuss the features of D-Stampede, the programming ease it affords, and its performance at a micro-level and at the level of a prototypical application.

Keywords: distributed programming, runtime system, synchronization, heterogeneity, ubiquitous computing.

1 Introduction

Marc Weiser envisioned *ubiquitous computing* as "computing that is an integral and invisible part of the way people live their lives" [1]. The explosive growth of technology, the falling price of computing and communication gear, and the permeation of technology into all aspects of our everyday living is giving credence that this vision will become a reality in the not too distant future. That the future of information technology will be dominated by invisible or pervasive computing is a belief that is being shared by several research groups.

Consider the following scenario:

John is sitting in his living room. He opens a connection to a virtual chat room and joins the discussion. Coordinated video and audio sensors capture John's appearance (including facial expressions and hand gestures) and speech in real-time. This information is transmitted across the network and used to reconstruct a virtual avatar of John. Each participant in the chat session sees and hears the avatars for the other participants, as if they were physically present. The avatars are created by a system of projectors and speakers. By projecting John's image onto a nearby wall and using spatial audio synthesis, his presence can be embedded in each participant's environment.

Complex ubiquitous computing applications, like the telepresence scenario described above, require the acquisition, processing, synthesis, and correlation (often *temporally*) of streaming data such as video and audio. Such applications usually span a multitude of devices that are *physically distributed* and *heterogeneous*. Different kinds of sensors (and data aggregators located near them) collect raw data and perhaps do limited processing such as filtering. However, extraction of higher order information content from such raw data requires significantly more processing power. For example, microphones may collect audio data, but higher order processing is needed for voice recognition. Thus there is a continuum of computation and communication resources as depicted in Figure 1. Also such applications are highly dynamic. For example, in a telepresence application participants may join and leave a chat session at different times.

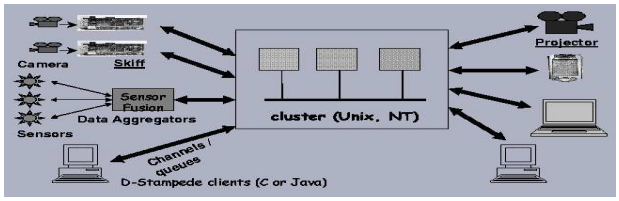

Figure 1: Octopus Hardware Model

The unique characteristics of this emerging class of distributed applications calls for novel solutions. In particular, there is a need for a distributed programming system that caters to the needs of such applications. Messaging layers such as MPI [2] and PVM [3] and middleware such as CORBA [4] and RMI [5] provide the basic transport and remote procedure call mechanisms needed in such distributed applications. There have been several language proposals for parallel and distributed computing such as Linda [6], Orca [7], and Cid [8]. Such languages provide fairly generic programming capabilities for data sharing and synchronization and do not offer any specific help for the common characteristics (such as temporal correlation of data from different streams) found in this application domain.

We have developed a novel distributed programming system,

*To appear in ICDCS 2002

†College Of Computing, Georgia Institute of Technology 801 Atlantic Drive, NW, Atlanta, GA 30332-0280, USA sameera, arnab, rama@cc.gatech.edu

called *D-Stampede*, that enables the coupling of networks of spatially distributed sensors and and data aggregators to clusters of compute servers. It has five main features:

- It is based on an "Octopus" architectural model (Figure 1) in which a central cluster of computing nodes (the "body") supports a network of sensors and data aggregators (the "tentacles").
- It supports a heterogeneous hardware continuum encompassing networked sensors, embedded data aggregators that connect to subsets of sensors, and backend computational clusters.
- It supports a heterogeneous programming environment wherein parts of the same application can be programmed in C and Java. All the parts have access to the same set of abstractions via a uniform set of API calls.
- It supports the sharing of streaming data across multiple nodes and address spaces. All aspects of the system, from the sharing primitives to an automatic garbage collection facility, are tailored to the properties of time-sequenced data that is a common trait of this application domain.
- It is designed for high performance, and supports both task and data parallelism in mapping target applications to the cluster infrastructure.

The D-Stampede system is currently operational and is being used for developing prototype ubiquitous computing applications such as telepresence in collaboration with researchers in computer vision and HCI at Georgia Tech. We have studied the performance of the D-Stampede system at two levels:

1. The overhead of using D-Stampede's high-level API compared to the basic message transport on which the system has been built.
2. The scalability at the application level of a workload (akin to distributed telepresence) implemented on top of D-Stampede.

Projects such as MIT Oxygen [9], Berkeley Endeavour [10], and OGI/Georgia Tech Infosphere [11] have high level objectives similar to ours, namely, to provide ubiquitous access to information. We differ in the specific research goals from those projects. The focus of the Oxygen project is to develop the fundamental technology for a ubiquitous computing fabric (from handhelds to powerful servers that may be implanted in the walls and ceilings) and address the issues of networking them and developing adaptive applications on top of them. The focus of the Endeavour project is to develop scalable services such as file systems on planetary scales on top of ubiquitous infrastructure with varied network connectivities. The focus of the Infosphere project is to devise middleware that will help the end user combat with the explosive growth of information on the internet (thanks to the world wide web), and allow the information infrastructure to scale as the use of the world wide web grows in volume and sophistication. There are a number of groups in industry and academia that are designing tiny integrated sensors that combine processing and communication in addition to sensing [12], and adaptive communication techniques in a network of sensors [13].

Our research goals are complementary to all of the above projects. Specifically, we wish to address the parallel/distributed programming problem engendered by the ubiquitous computing environment. We begin by discussing the requirements of ubiquitous computing applications in Section 2. We present the system architecture of D-Stampede and its implementation in Section 3. A flavor of programming experience with D-Stampede is presented in Section 4. We discuss performance of D-Stampede in section 5, and conclude in section 6.

2 Requirements

We enumerate the distributed programming requirements for ubiquitous computing applications:

1. *Stream Handling:* Today video and audio sensors are common. Consider gesture and speech as inputs for a ubiquitous application. A gesture is a sequence of images, and speech is a sequence of audio samples. The import of a word would depend on the associated gesture. Each stream is a sequence of some basic basic element (e.g. a frame of pixels). As sensor based computing becomes more prevalent we expect there to be a preponderance of stream data. Thus there needs to be be efficient support for streams.
2. *Temporal Indexing and Correlation:* Datasets from different sources may need to be combined, correlating them temporally. For example, a stereo vision application would combine images captured at the same time from two different camera sensors, and stereo audio combines data from two or more microphones. Other analyzers may work multimodally, *e.g.,* by combining vision, audio, gestures and and other sensor inputs. These requirements suggest that application development would benefit considerably if the programming system offered some support for indexing data by time.
3. *Distributed Heterogeneous Components:* The very nature of the application suggests that the components of the system are going to be distributed and heterogeneous. The heterogeneity may be both at the level of the hardware as well as the system software that runs on the components. For *e.g.,* a sensor may have an integrated Java virtual machine associated with it and JVM may be the only interface exported by the device to the application programmer. Further the application programmer may want a choice in the language used for different components of the same application. For *e.g.,* some compute intensive algorithms (such as a color tracker) may be coded in C but it may be more expedient to code some other part of the application (such as a video display perhaps) in a language like Java.
4. *Real time Guarantees:* Due to the interactive nature, certain events in the application may need soft real-time guarantees.

For *e.g.,* in telepresence timely visual and audio feedback to the participants is essential.

5. *Dynamic Start/Stop:* There should be a natural way for components of the application to join and leave. For *e.g.,*, in a telepresence application a camera sensor may come alive when a participant joins a chat session and may go offline when the participant signs off.

6. *Plumbing:* Ubiquitous computing applications consist of several different computational modules that are connected to one another in complex fashion. Further, such plumbing may not always be statically definable. Clearly, the application structure is going to mirror the Octopus hardware (Figure 1 model that we alluded to earlier). The programming system should allow for intuitive, efficient, and flexible way of dynamically instantiating such complex pipelines of computational modules.

7. *Dynamic Resource Management:* There are two aspects to resource management in the context of such applications: Firstly, it may be necessary to dynamically provide more processing power for certain aspects of the application than others. For *e.g.,* analyzing an image for objects of interest may require exploiting data parallelism. Secondly, efficient management and recycling of memory buffers that act as conduits between the computational modules is crucial for application scalability. These applications are continuous in nature. Sensors continually produce data which is passed on to subsequent stages for processing and higher order inferencing. To ease the programming burden the runtime system should facilitate automatic reclamation of memory resources based on user level hints.

There are other aspects of this application context that require support from the programming infrastructure as well. These include *High Availability*, and *Security and Privacy*. Due to the continuous nature of these applications, temporary failures of individual components should not lead to failure of the entire application. Further, with sensors in the environment unbeknownst to the people who may occupy these spaces, there is a need to address security and privacy concerns in ubiquitous computing applications. These issues are outside the scope of this paper.

3 The D-Stampede Programming System

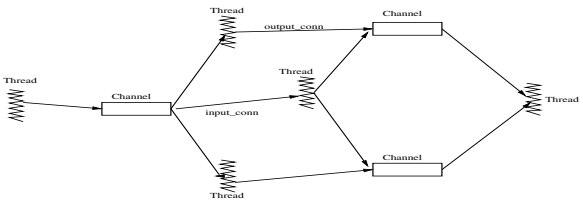

Figure 2: Computational Model : A dynamic Thread Channel Graph

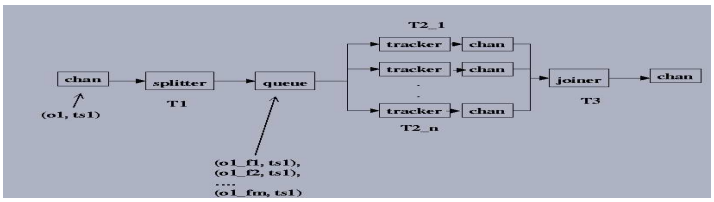

Figure 3: An example of Task Data Parallelism : o1_f1, o1_f2,.., o1_fm are fragments of the same frame o1 with timestamp ts1. These frame-fragments are analyzed in parallel by trackers. The tracker outputs for the same timestamp but different frame-fragments are combined by the joiner thread.

The computational model supported by D-Stampede is pictorially presented by the thread-channel graph in Figure 2. The model captures a huge class of inherently distributed applications that are emerging in the context of ubiquitous computing. The threads map to computing devices ranging from small to high-end processing elements scattered in the Octopus hardware model (Figure 1). Channels serve as the application level conduits among the threads, specifically tuned to handle time-sequenced streaming data. D-Stampede provides APIs for the dynamic creation of threads and channels, and for the dynamic establishment (and removal) of the plumbing among them.

3.1 Architecture

The D-Stampede architecture has the following main components: *Channels*, *Queues*, *Threads*, *Garbage Collection*, *Handler Functions*, *Real-time Synchrony*, *Nameserver*, and support for *Heterogeneity*.

- *Threads, Channels, and Queues:* D-Stampede provides a uniform set of computational abstractions across the entire hardware continuum: threads, channels, and queues. Stampede threads are POSIX-like and can be created in different protection domains (address spaces) for memory isolation purposes [14] . Channels and queues are system-wide unique names, and serve as containers for storing *time-sequenced* data items produced by threads. They facilitate inter-thread communication and synchronization regardless of the physical location of the threads, channels, and queues. A thread (dynamically) 'connects' to a channel (or a queue) for *input* and/or *output*. Once connected, a thread can do I/O (in the form get/put items) on the channel (or queue). The items represent some application-defined chunking of stream data (for e.g. frames of video, audio samples, etc.). The timestamps associated with an item in a channel (or queue) is user defined (for e.g. timestamps may be the frame number of video generated in a telepresence application). The collection of time-sequenced data in the channels and queues is referred to as *space-time memory* [15].

  While the channel allows random access by a thread for items of interest (based on the timestamp value associated with an item), a queue, as the name suggests allows FIFO access to items contained in it. The queue abstraction is primarily

designed to exploit any data parallelism in an application. For example, if it is desired to analyze a given frame of video for objects of interest, then the frame can be partitioned into frame-fragments (all having the same timestamp) and placed in a queue by a splitter thread. A distinct thread can analyze each frame-fragment for objects of interest. A joiner thread can then stitch together the composite analyzed outputs. This is pictorially shown in Figure 3.

At a conceptual level, a D-Stampede computation with threads, channels, and queues is akin to a distributed computation specified by a set of processes connected by sockets. This conceptual equivalence makes it an easy transition from socket-based distributed programming to channel-based D-Stampede programming. The power of D-Stampede is the ability afforded to the application for reasoning about program behavior based on time (an important feature in interactive distributed applications such as telepresence), and temporal correlation among data generated by different sources (e.g. audio and video coming from an avatar in telepresence).

- *Garbage Collection:*

  D-Stampede aids the development of highly dynamic applications. The runtime mechanisms in D-Stampede mirror the dynamism in the applications. The human body uses *selective attention* as a means of filtering out inputs from unrelated sensors while performing a specific bodily function (take vision for example). In a similar vein, D-Stampede facilitates selective attention at two levels: first by allowing a thread to dynamically choose the set of channels and queues it wants to perform I/O on, and second by using timestamps as a filtering mechanism. API calls in D-Stampede facilitate a given thread to indicate that an item (or a set of items) in a channel or a queue is garbage so far as it is concerned. Using this per-thread knowledge, D-Stampede automatically performs distributed garbage collection [16] of timestamps (i.e. items with such timestamps) that are of no interest to any thread in the D-Stampede computation.

- *Handler Functions:* D-Stampede allows association of *handler functions* with channels and queues for applying a user-defined function on an item in a channel (or a queue). The handler functions come in handy in a variety of situations. For e.g., if an item (which may be a complex user-defined data structure) has to be transported across address spaces (and/or machine boundaries), the user can define serialization and de-serialization handlers that D-Stampede will invoke as necessary to perform the API calls (such as a get from a thread in a remote address space). Similarly, once an item is determined to be garbage by the runtime system, a user-defined handler can be invoked to garbage collect memory buffers in user space associated with that item.

- *Real-time Synchrony:*

  The timestamp associated with an item is merely an indexing system for data items, and does not in itself have any direct connection with real time. For pacing a thread relative to real time, D-Stampede provides an API for loose temporal synchrony that is borrowed from the Beehive system [17]. Essentially, a thread can declare real time "ticks" at which it will re-synchronize with real time, along with a tolerance and an exception handler. As the thread executes, after each "tick", it performs a D-Stampede call attempting to synchronize with real time. If it is early, the thread waits until that synchrony is achieved. It if is late by more than the specified tolerance, D-Stampede calls the thread's registered exception handler which can attempt to recover from this slippage. Using these mechanisms, for example, a camera in a telepresence application can pace itself to grab images and put them into its output channel at 30 frames per second, using absolute frame numbers as timestamps.

- *Name Server:* D-Stampede infrastructure includes a name server. Application threads can register (and un-register) all pertinent information (such as names of channels and queues, as well as their intended use in the application) with this name server. Any new thread that starts up in the application anywhere in the entire network of the Octopus model can query this name server to determine resources of interest that it may want to connect to in the computation model of Figure 2. This facilitates the dynamic start/stop feature we alluded to in Section 2.

- *Heterogeneity:* We made a conscious decision to develop D-Stampede as a runtime library on top of standard operating systems. Accordingly, we have architected the D-Stampede system as "client" libraries on the end devices (i.e. tentacles of the Octopus) and a "server" library on the cluster (i.e. the head of the Octopus). This organization has no bearing on the generality of the computational abstractions provided by D-Stampede. The API calls of D-Stampede are available to a thread regardless of where it is executing. D-Stampede accommodates heterogeneity of components in two ways. Firstly, the server library of D-Stampede has been ported to the following platforms: DEC Alpha-Tru64, x86-Linux, x86-Solaris, and x86-NT. Secondly, D-Stampede supports *language* heterogeneity on the end devices (i.e. the clients), which can be programmed in either C or Java. A D-Stampede application can have some parts written in C and some parts written in Java sharing the same data abstractions. We elaborate on the implementation details in Section 3.2.

To give a flavor of programming with D-Stampede, we present below pseudocodes for a producer thread and a consumer thread.

```
/* Producer Thread */
connect( Channel, output mode );
for ( timestamp = start_time; timestamp < end_time;
timestamp++ )
put_item ( Channel, timestamp , item);
/*  Consumer Thread    */
connect( Channel, input mode );
for ( timestamp = start_time; timestamp < end_time;
timestamp++ )
get_item ( Channel, timestamp, &item_buffer );
/* use_item */
consume_item (Channel, timestamp); /* signal garbage */
```

Detailed discussion of the D-Stampede API is beyond the scope of this paper [1].

Suffice it to say that programming with D-Stampede requires thread and channel creation, thread-channel connection setup, and the actual I/O on the connections. The system provides a rich set of high level APIs that take care of the buffer management, synchronization, and garbage collection necessities of the application, allowing the programmer to concentrate on the specifics of the application itself.

3.2 Implementation

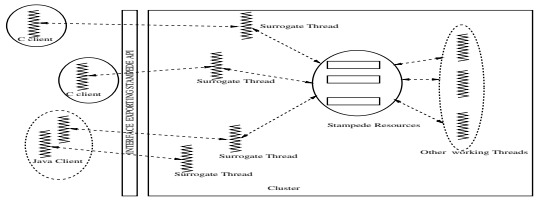

Figure 4: D-Stampede Implementation

D-Stampede is implemented as a runtime library that has two parts, a *server* written in C running on a high performance cluster, and *clients* (written in C or Java) that can run anywhere in the distributed system. Figure 4 shows the organization. Though the programming model is uniform and does not distinguish between client and server, this dichotomy exists for both historical reason (Stampede was originally developed as a cluster computing system) and a reflection of the Octopus analogy we alluded to earlier (Figure 1). The distributed end points (clients) are usually connected to sensors and in general serve as the capture and access points in the environment, while heavy duty computation (such as tracking and image analysis) is performed on the cluster (server).

3.2.1 Client Library

The D-Stampede APIs are exported to the distributed end points in a manner analogous to exporting a procedure call using an RPC interface [18]. There are client libraries available for both C and Java. The Java client library encapsulates the D-Stampede APIs as objects. The application level D-Stampede programs running on the end devices (i.e. clients) can be written in C or Java and they coexist as parts of a single application.

A TCP/IP socket is used as the transport for communication between the client and the server libraries. The Java client library uses our own data representation to perform the marshalling and unmarshalling of the arguments, while the C client library uses XDR [19].

3.2.2 Server Library

The server library is implemented on top of a message-passing substrate called CLF, a low level packet transport layer developed at Compaq CRL. CLF provides reliable, ordered point-to-point packet transport between the D-Stampede address spaces within the cluster, with the illusion of an infinite packet queue. It exploits shared memory within an SMP, and any available network between the nodes of the cluster, including Digital Memory Channel [20], Myrinet [21], and if none of these are available, UDP over a LAN.

There is a *listener* thread on the cluster (part of the server library) that listens to new end devices joining a D-Stampede computation. Upon joining, a specific *surrogate* thread (see Figure 4) is created on the cluster on behalf of the new end device. All subsequent D-Stampede calls from this end device are fielded and carried out by this specific surrogate thread. State information pertaining to an end device is maintained by the server library via the associated surrogate thread. The surrogate thread ceases to exist when the end device goes away. The creation/annihilation of this surrogate thread on the cluster mirrors the joining and leaving of an end device.

Garbage collection is performed on the cluster concurrent with application execution. The surrogate threads participate in garbage collection on behalf of the end devices. The server library notifies the client libraries for any storage reclamation that has to happen on an end device as a result of garbage collection on the cluster.

3.2.3 Supporting Heterogeneity

Java clients can run on any end device that has a JVM. The server library (which is in C) and the C client library have been ported to the following platforms: DEC Alpha-Tru64 Unix, x86-Linux, x86-Solaris, and x86-NT. The implementation supports combinations of 32-bit and 64-bit platforms for the end devices and the cluster. Any combination of end devices and a cluster platform can host a single D-Stampede application.

3.2.4 Supporting Handler Functions for End Devices

As we mentioned earlier (see Section 3.1), D-Stampede allows user-defined actions to be performed on an item in a channel or a queue via the handler function. Garbage collection is a good example for use of this mechanism. Upon the D-Stampede runtime determining that an item in a channel is garbage, it calls the user-defined handler to free up any memory in user space associated with that item. While this is quite naturally implemented for D-Stampede threads running in the cluster, special handling is needed for the end devices. When requested by an end device to install a handler, its surrogate installs a generic handler function on the cluster. When this generic handler is invoked by the D-Stampede runtime, it collects the information on behalf of the end device and communicates it to the end device at an opportune time (for e.g. when the next D-Stampede API call comes from the end device).

3.3 Limitations in the Current System

There are a couple of limitations in the current D-Stampede system.

[1] A header file that gives the D-Stampede API may be found at http://www.cc.gatech.edu/ rama/stampede/api_h.txt

- Currently, there is no support in D-Stampede for handling failures. For e.g., if an end device does not cleanly leave an application, perhaps due to a hard failure or a client program crash, it will leave its surrogate on the cluster in an indeterminate state.

- Currently, the implementation does not support a given D-Stampede application spanning multiple clusters. While there can be any number of end devices, there can only be one cluster involved in an application. Further, the server library assumes a homogeneous cluster (i.e all the cluster nodes run the same operating system on the same processor architecture).

4 Programming using D-Stampede

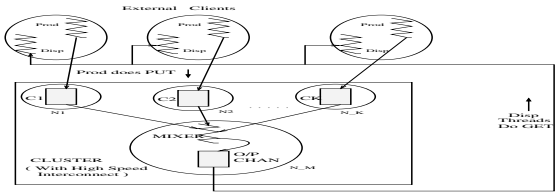

Figure 5: Structure of Video Conferencing Application

Figure 6: Sample Display Output to Each Participant

Consider a video conferencing application. Conceptually, this application involves combining streams of audio and video data from multiple participants and sending the composite streams back out to the participants. To keep the discussion simple, we will only consider video streams. An implementation of this application using D-Stampede will have the following components:

- A server program on the cluster that has a set of channels (one per end device) for storing the video streams; a mixer thread that takes corresponding timestamped frames from these channels to create a composite video output; and a channel for placing the composite video output of the mixer thread.

- A camera and a display as end devices for each participant. A client program on each end device that is comprised of a producer thread that 'puts' its timestamped video stream on its assigned channel; and a display thread that 'gets' the composite video and displays it to the participant.

Figure 5 pictorially shows the structure of this application. Figure 6 shows a sample display (a composite of 4 camera inputs) of this application as seen by each participant.

When the server program is started on the cluster the following events happen:

1) The server program creates multiple address spaces $N_1, N_2 \ldots N_k$ in the cluster.
2) The server library spawns a listener thread in each address space.
3) The server program creates address space N_M, where a mixer thread is spawned, and a channel C_o is created.
4) The id of channel C_j created by j-th client $(1 \leq j \leq m)$ in N_i $(1 \leq i \leq k)$ is made available to N_M via the nameserver.
5) The id of channel C_o in N_M is made available to each client via the nameserver.

The mixer thread does the following:

1) Create input connections to channels C_j $(1 \leq j \leq m)$.
2) Create output connection to the channel C_o.
3) Get correspondingly timestamped items (images) from each C_i.
4) Create the composite item (image).
5) Put composite item on channel C_o.

When the j-th client program is started on an end device, the following events happen:

1) The client library implicitly communicates with a listener thread in one of the address spaces N_i $(1 \leq i \leq k)$.
2) The client program creates a channel C_j in address space N_i.
3) The client program starts a producer thread which creates an output connection to channel C_j and puts images in C_j.
4) The client program starts a display thread, which creates an input connection to channel C_o and gets composite images from C_o.

D-Stampede eases the task of developing such an application by providing high level computational abstractions. As can be seen from the above example, the features in D-Stampede – such as specifying an arbitrary number of address spaces, creating any number of threads in different address spaces, creating system-wide unique channels, transporting stream data that can be chunked in any arbitrary user-defined manner, timestamping the data items, and time-correlating different data streams – come in very handy for developing such interactive stream-oriented applications.

5 Performance

We have carried out an experimental study to evaluate the performance of the D-Stampede system. We have measured the performance of the system at two levels. First at the micro-level to determine the latency of D-Stampede operations. Second at the level of a video-conferencing application that we discussed in the previous section. The hardware setup for the experimental study is as follows: A cluster consisting of 17 eight-way SMPs interconnected by Gigabit Ethernet. Each processor is a 550MHz Pentium III Xeon. Each node in the cluster has 4GB RAM and 18GB SCSI disk. The nodes run RedHat linux 7.1. Note that even in experiments that involve the client libraries for C and Java, one of the cluster node acts as an end device. This experimental setup ensures that the measurements reflect the impact of the D-Stampede software libraries and are not perturbed by the vagaries in the end device hardware capabilities.

5.1 Micro Measurements

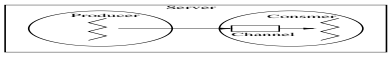

Figure 7: Experiment 1

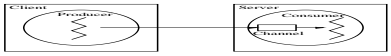

Figure 8: Experiment 2, Configuration 1

Figure 9: Experiment 2, Configuration 2

The micro-level measurements compare an end-to-end data exchange between two D-Stampede threads against a similar data exchange using the underlying messaging layer. The intent is to show that the overhead incurred for the high-level data abstractions in D-Stampede is minimal compared to raw messaging. For raw messaging, this data exchange amounts to a send-receive combination. A put-get combination is the logical equivalent for D-Stampede. We conduct three experiments, each involving a pair of producer-consumer threads. In each experiment we compare different configurations of producer-consumer pair performing a data exchange in D-Stampede against a commensurate send-receive based data exchange. In the graphs that correspond to these experiments (Figures 11, 12, and 13), the message size (in bytes) is plotted along the X-axis, and the latency (in microseconds) for the data exchange is plotted along the Y-axis.

The readings shown are for data sizes ranging from 1000 to 60000 bytes, in steps of 1000 bytes. We restricted our readings to 60000 bytes because UDP does not allow messages greater than 64 KB.

Experiment 1: In this experiment (pictorially shown in Figure 7) the producer and consumer threads are on different nodes (and therefore different address spaces) *within* the cluster. The channel used for communication is located in the consumer's address space. The producer puts items on the channel. The consumer gets items from the channel. We orchestrate the experiment such that the put and get do not overlap. Latency is measured as the sum of the put and get operations.

This D-Stampede configuration is compared against two other alternative scenarios. One alternative uses UDP for communication and the other TCP/IP between the producer-consumer pair. Within the cluster, D-Stampede uses CLF, a reliable packet transport layer built on top of UDP. CLF does not have the connection management overheads of TCP/IP, and hence the two alternative scenarios being compared against D-Stampede. To ensure that the send and the receive do not overlap for the two alternative scenarios, we do the following: The producer sends a message and the consumer receives it. The consumer sends a message and the producer receives it. The exchange latency is assumed to be half the time taken for this cycle. The producer-consumer programs for this experiment are all written in C. The performance results are shown in Figure 11.

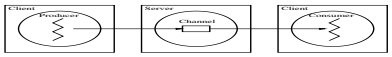

Figure 10: Experiment 2, Configuration 3

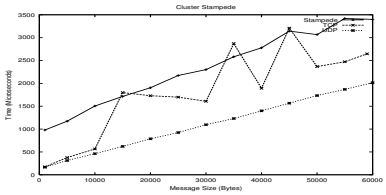

Figure 11: Intra-Cluster Results (Experiment 1)

As can be seen, data exchange using D-Stampede adds an overhead (that ranges from 700 microseconds at 10 KB payload to 1200 microseconds at 60 KB payload) compared to the UDP alternative. The overhead compared to the TCP/IP alternative is much less (starts from around 700 microseconds at 10 KB and with the increase of message size falls to 400 microseconds at 60 KB). The TCP/IP based producer-consumer measurements have some spikes that are due to the inherent congestion control properties of TCP/IP protocol. Overall, the overhead incurred for D-Stampede is fairly low at reasonably high payloads (less than 2X compared to UDP; at best almost the same or better than TCP/IP and at worst within 1.5X compared to TCP/IP).

Experiment 2: This experiment involves the use of the C client library of D-Stampede. The producer thread runs on an end device as a client program. There are three configurations used in this experiment, each differing in the location of the consumer thread.

- **Configuration 1:** As shown in Figure 8, the consumer thread is co-located with the channel on the cluster. This configuration involves one end device to cluster network traversal. The get operation is local to the cluster node due to the co-location of the channel. Recall that the client library uses TCP/IP to communicate with the server library. Thus, this configuration shows the exact overhead that D-Stampede runtime adds to TCP/IP. For example, for a data

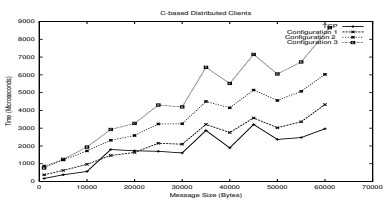

Figure 12: C based End Device and Cluster Results (Experiment 2)

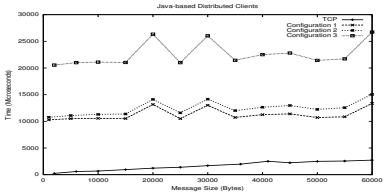

Figure 13: Java based End Device and Cluster Results (Experiment 3)

size of 55000 bytes, TCP/IP latency is 2500 μs, and D-Stampede latency is 3300 μs.

- **Configuration 2:** As shown in Figure 9, the consumer thread is located on the cluster. The channel is in a different address space from that of the consumer. This configuration involves one end device to cluster network traversal (for the put operation) and one intra-cluster network traversal (for the get operation). Thus, this configuration is expected to have more overhead, compared to the previous configuration. For 55000 bytes payload, the D-Stampede latency is around 5000 μs.

- **Configuration 3:** As shown in Figure 10, the consumer thread is located on an end device. This configuration involves two end device to cluster network traversals, one for each of the put and get operations. Thus, this configuration has the maximum overhead. For 55000 bytes payload, the D-Stampede latency is around 6100 μs

Since all the D-Stampede configurations used in this experiment involve the client library (which uses TCP/IP for communicating with the server library), we use a TCP/IP based producer-consumer pair written in C for comparison with the D-Stampede results. As before, we orchestrate the experiment to ensure that the producer and consumer do not overlap in their communication. The results for this experiment are shown in Figure 12. As expected, the shape of the D-Stampede curves track the TCP/IP curve for all the configurations. For configuration 1 (wherein there is only one end device to cluster network traversal) the D-Stampede runtime overhead over TCP/IP is fairly nominal (at best less than 12%).

Experiment 3: This experiment is the almost the same as the previous one with the only difference that the Java client library is used for D-Stampede, and the TCP/IP based producer-consumer program for comparison is also written in Java. The results for this experiment are shown in figure 13 For a payload of 55000 bytes, the D-Stampede latency is approximately 11000 μs for configuration 1 (corresponds to Figure 8), approximately 12600 μs for configuration 2 (corresponds to Figure 9), and approximately 21700 μs for configuration 3 (corresponds to Figure 10).

The results of the three experiments are summarized below:

Result 1: For D-Stampede, intra-cluster data exchange (Figure 11) performs slightly better than C-based end device to cluster data exchange (Figure 12), which in turn performs better than Java-based end device to cluster data exchange (Figure 13). For example, the latency for a data size of 35000 bytes, is 2580 μs in the first case, 3200 μs in the second and 10700 μs for the third case. Since the hardware is the same for all these data exchanges, the disparity is simply due to the software. As we mentioned earlier, D-Stampede uses CLF for intra-cluster communication, which has less overhead compared to TCP/IP.

Result 2: For TCP/IP based data exchange, the C producer-consumer program results (Figure 12), and Java producer-consumer program results (Figure 13) are similar. However, the D-Stampede data exchange is much better in C compared to Java. In C marshalling and unmarshalling arguments involve mostly pointer manipulation, while in Java they involve construction of objects. Hence the disparity between the two. A potential for further experimentation is using JNI to wrap the C functions for marshalling and unmarshalling and using them for the Java clients as well.

5.2 Application Level Measurements

We use the application described in Section 4 for this study. We abstract out the camera and display from the application to make the study a controlled experiment for evaluating scalability with respect to the D-Stampede system. The producer thread in the client program reads a "virtual" camera (a memory buffer) and sends it to the server program continuously. Similarly the display thread in the client program simply absorbs the composite output from the mixer without really displaying it. This structure allows us to stress the communication infrastructure of D-Stampede at the maximum possible rate the application pipeline can generate video without interfacing to any external I/O devices. Sustained *frame rate* is the performance metric of interest in this application.

We have developed three versions of this application. The first version uses Unix TCP/IP socket for communication between the client programs and the server program. The mixer (a single thread) obtains images from each client one after the other, generates the composite, and sends it to the clients one after the other. The second version is almost the same as the first version, except that it uses a D-Stampede channel between each client program and the server program (i.e. in Figure 5, the mixer in address space N_M is single threaded). The third version has a multi-threaded mixer using D-Stampede channels (i.e. the mixer in address space N_M is multi-threaded in Figure 5). There is one thread in the mixer for each client program. Each thread obtains an input image from the associated client, and performs its part of the composite image generation. Once the image is fully constructed, it is placed in the channel by a designated thread in the mixer for the client programs to pick up for display.

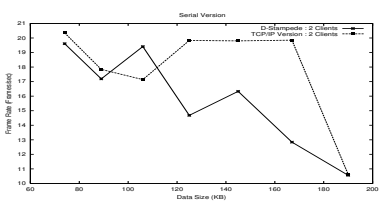

Figure 14: Single Threaded Version performance

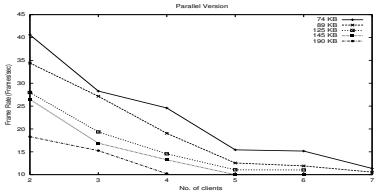

Figure 15: Multi-threaded Version performance

Data size (KB)	Delivered Bandwidth (MBps)					
	Number of Clients					
	2	3	4	5	6	7
74	11	18	28	28	39	42
89	11	21	26	28	40	46
125	13	20	29	36	48	
145	14	21	29	35	50	
190	13	25	30			

Table 1: Delivered Bandwidth (MBps) as a function of data size and number of clients.

In Figures 14 and 15, we have only shown readings when the sustained frame rate at the display thread is higher than 10 frames/sec. We feel that any lesser frame rate at the display thread than this threshold would be unacceptable for a video conferencing application. Figure 14 shows the performance of the first two versions (both single threaded), for two clients[2], and for image sizes from 74 KB to 190 KB per client. This figure plots the sustained frame rate (on the Y-axis) as a function of the image data size (X-axis). As can be seen from the figure, the performance of the socket version[3] and the D-Stampede channel version are comparable for the most part. For example, for a data size of 110 kb, they both deliver 18 frames/second sustained frame rate. This exercise proved two things. 1) Due to the complexity of this application, writing it using sockets required much more effort compared to D-Stampede. 2) The performance of D-Stampede version is comparable to the socket version.

In Figure 15, the performance of the multi-threaded version is shown by plotting the sustained frame rate (Y-axis) as a function of the number of participants (X-axis). Each line in the graph is for a different client image size. Therefore, for an instance of the application with K clients, the display thread at each client receives a frame K times bigger than the client image size. The sustained frame rates measured at the different participants varied in a narrow band for each client image size. We chose the slowest display frame rate among the participants as the representative for each client image size.

Comparing Figures 14 and 15, we see that the multithreaded version performs better than the single threaded versions. For a client image size of 74KB, the single threaded version delivers approximately 20 frames/sec. In contrast the multithreaded version delivers approximately 40 frames/second. Clearly, thread parallelism in the mixer helps to boost the sustained frame rate seen at the display threads.

Despite the thread parallelism in the mixer, it can be seen from Figure 15 that the sustained frame rate achieved at a display thread is a function of the number of participating clients, the per client image data size, and the available network bandwidth between the cluster and the end devices. For an image size of 74 KB, we see a frame rate of around 40 frames/sec for 2 clients, which drops to approximately 30 frames/sec for 3 clients. Similarly, for 2 clients, with an 89 KB image size, we get approximately 34 frames/sec, which drops to approximately 27 frames/sec for 125 KB image size.

In this version of the application, all the threads of the mixer run in one node (an 8-way SMP) of the cluster. Thus all the client display threads feed out of this one node to get the composite image data from the channel. The available network bandwidth at this cluster node is a complex function of the simultaneity of the requests from the display threads, the bandwidth of the memory subsystem, and other vagaries such as thread scheduling by the operating system. Since mixing is the most compute intensive operation in this application pipeline, it is highly likely that the requests from the display threads for the composite image are serviced simultaneously. Therefore, for K clients, with a per client image size of S, and a frame rate F, the required bandwidth at this cluster node is K^2SF. This is because each client receives a composite of size KS, and there are K clients. Table 1 summarizes the actual delivered bandwidth from this cluster node for the various client image sizes and number of clients. This table is derived from the measurements which are plotted in Figure 15. From the table, it can be seen that the sustained frame rate falls below the 10 frames/sec threshold when the required bandwidth exceeds 50 MBps, suggesting that this is perhaps the maximum available network bandwidth out of the cluster node. This situation happens with 5 clients when the image size is 190KB, and 7 clients for the other lesser image sizes. We point this out to emphasize that the observed limit to the scalability is the application structure and not any limitation in the D-Stampede implementation.

6 Conclusion

D-Stampede is a programming system that is designed for supporting interactive stream-based distributed ubiquitous computing applications. The key features that D-Stampede provides for this application domain include indexing data streams temporally, correlating different data streams temporally, performing automatic distributed garbage collection of unnecessary stream data, supporting high performance by exploiting hardware parallelism where available, supporting platform and language heterogeneity, and dealing with the dynamism of such applications. We have illustrated the ease of developing ubiquitous computing applications on D-Stampede. Through micro-level and application-level measurements, we have shown that the programming ease attained with D-Stampede does not adversely affect performance.

[2]The other configurations of the single threaded version that met our threshold are: a) 3 participants with images sizes 74 KB, 89 KB and 106 KB, b) 4 participants with image size 74 KB. No configuration with 5 or more participants met our threshold of 10 frames/sec.

[3]We chose to implement the application using TCP/IP sockets for comparison with D-Stampede channels since other alternatives such as Java RMI [5] or CORBA [4] have significantly higher overhead.

D-Stampede is currently available as a runtime library that runs on a variety of platforms, and is being used by researchers in computer vision and HCI at Georgia Tech as a development platform for next generation media applications such as telepresence. There are a number of system issues that we are currently working on. The first has to do with generalizing the hardware model for which D-Stampede is intended. In particular, we would like to extend the D-Stampede system to support multiple heterogeneous clusters connected to a plethora of end devices participating in the same D-Stampede application. Extending the selective attention capability of D-Stampede to perform user defined filtering operations is another avenue of future research. A third area of future research is dealing with failures, both towards developing a computational model as well as efficient runtime support for the model.

Acknowledgments

A number of people have contributed to the D-Stampede project. Rishiyur Nikhil, Jim Rehg, Kath Knobe, and Nissim Harel contributed to the space-time memory abstraction that is at the heart of D-Stampede. In addition, Bert Halstead, Chris Jeorg, Leonidas Kontothanassis, and Jamey Hicks contributed during the early stages of the Stampede project. Dave Panariti developed a version of CLF that runs transparently on Alpha Tru64 and Microsoft NT. All the members of the "ubiquitous presence" group at Georgia Tech continue to contribute to the D-Stampede project. Bikash Agarwalla, Matt Wolenetz, Hasnain Mandviwala, Phil Hutto, Durga Devi Mannaru, Namgeun Jeong, Yavor Angelov, and Ansley Post deserve special mention. Russ Keldorph and Anand Lakshminarayanan developed an audio and video meeting application on D-Stampede.

References

[1] Mark Weiser. *The Computer for the 21st Century.* In Scientific American, September 1991.

[2] MPI Forum. *MPI: Message-Passing Interface Standard.* Mar 1994.

[3] V. S. Sunderam. *PVM: A framework for parallel distributed computing.* Concurrency: Practice and Experience, Vol. 2(4), p: 315-339, Dec. 1990.

[4] Object Management Group. *CORBA* www.corba.org

[5] Sun Microsystems. *Java RMI* www.java.sun.com

[6] S. Ahuja, N. Carriero and D. Gelernter. *Linda and friends.* IEEE Computer, Vol. 19(8), p: 26-34, Aug 86.

[7] H.E. Bal, M.F. Kaashoek and A.S. Tanenbaum. *Orca: A Language for Parallel Programming of Distributed Systems.* IEEE Transactions on Software Engineering, Vol. 18(3), p: 190-205, Mar 90.

[8] R. S. Nikhil. *Cid: A parallel shared-memory C for distributed memory machines.* Seventh International Workshop on Languages and Compilers for Parallel Computing, Ithaca, Aug. 1994 Springer-Verlog, p: 377-390.

[9] *Oxygen.* http://oxygen.lcs.mit.edu

[10] R. H. Katz. *The Endeavour Expedition: Charting the fluid information utility.* http://endeavour.cs.berkeley.edu

[11] *Infosphere* www.cc.gatech.edu/projects/infosphere

[12] J. Hill, R. Szewczyk, A. Woo, D. Culler, S. Hollar, and K. Pister. *System architecture directions for networked sensors.* The Nineth International Conference on Architectural Support for Programming Languages and Operating Systems, Nov 00 p: 93-104.

[13] J. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, D. Ganesan. *Building Efficient Wireless Sensor Networks with Low-Level Naming.* The Eigth ACM Symposium on Operating Systems Principles, Oct 01.

[14] R. S. Nikhil, U. Ramachandran, J. M. Rehg, R. H. Halstead, Jr., C. F. Joerg and L. Kontothanassis. *Stampede: A programming system for emerging scalable interactive multimedia applications.* The 11th International Workshop on Languages and Compilers for Parallel Computing, 98.

[15] U. Ramachandran, R. S. Nikhil, N. Harel, J. M. Rehg and K. Knobe. *Space-Time Memory: A Parallel Programming Abstraction for Interactive Multimedia Applications.* The Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, May 99.

[16] R. S. Nikhil, and U. Ramachandran. *Garbage Collection of Timestamped Data in Stampede.* Principles of Distributed Computing, Jul 00 .

[17] A. Singla, U. Ramachandran and J. Hodgins, *Temporal Notions of Synchronization and Consistency in Beehive*, The Nineth Annual ACM Symposium on Parallel Algorithms and Architectures, Jun 97.

[18] Andrew D. Birrell and Bruce J. Nelson. *Implementing Remote Procedure Calls* ACM Transactions on Computer Systems, Vol. 2(1), Feb 84, p: 39-59.

[19] R. Srinivasan. *Rfc 1832: Xdr: External data representation standard.* Aug 95.

[20] R. Gillett, M. Collins, and D. Pimm. *Overview of network memory channel for PCI.* The IEEE Spring COMPCON '96, Feb. 96.

[21] N. J. Boden, D. Cohen, R. E. Felderman, A.E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. *Myrinet: A Gigabit-per-Second Local Area Network.* IEEE Micro, February 95 p: 29–36.