

# Stampede<sup>RT</sup>: Programming Abstractions for Live Streaming Applications

David Hilley

Umakishore Ramachandran

*College of Computing, Georgia Institute of Technology*  
{*davidhi, rama*}@cc.gatech.edu

## Abstract

*We present Stampede<sup>RT</sup>, middleware designed to provide a natural programming model appropriate for live streaming applications. Such applications require pervasive access to multiple streaming data sources for distributed online analysis. One motivating example is a distributed robotics application which analyzes live camera feeds for control and planning. Most existing middlewares for streaming data focus on media streams and low-level transport characteristics such as delivery latency and efficient transfer, but do not define a programming model to succinctly express applications that manipulate and analyze the streaming content. Stampede<sup>RT</sup> provides for straightforward transport and manipulation of temporally-ordered data streams, enabling simple synchronization and correlation of data sources. We present an abstract programming model to support the aforementioned class of applications and then describe a concrete realization of the model as a distributed middleware architecture. We also evaluate our implementation of the architecture and present several motivating applications Stampede<sup>RT</sup> is designed to support.*

## 1. Introduction

One exciting class of distributed applications is based on the concurrent streaming and real-time analysis of media streams. Typically, such applications can be depicted as coarse-grain dataflow graphs of computationally-intensive processing modules dealing with multiple continuous data streams. Stampede<sup>RT</sup> is a distributed programming system designed to support such applications. While the aforementioned class of pervasive applications, termed **live streaming applications**, are becoming increasingly common – and important – traditional distributed programming infrastructures typically provide less than ideal programming models for supporting such applications. One such motivating example is a distributed robotics application called LAGR, which performs vision-based analysis on live camera feeds for control and planning. These applications have several basic characteristics: i) pervasive access to multiple live streaming data sources, like sensor feeds and broadcast media; ii) the need for online, intensive distributed analysis and correlation of temporally-ordered data; and iii) and time-based stream synchronization. Stampede<sup>RT</sup> provides a programming model directly supporting these requirements.

Most media middleware systems providing support for distributed streaming media focus on performance characteristics, as well as low-level details such as transport features and multicasting. In addition, a large percentage of streaming middleware systems are solely concerned with streaming for playback, and not online analysis. In fact, very few systems focus on providing natural abstractions to support applications with extensive stream-based manipulation. By defining abstractions with a transport-level recognition of time, Stampede<sup>RT</sup> provides a programming model convenient for expressing live streaming applications.

In a nutshell, Stampede<sup>RT</sup>'s programming model can be thought of as distributed data structures for streaming data. Threads of computation share queue-like data structures called **channels**, which are the main abstraction for transport of data streams and distributed communication. We define a “stream” as any continuous feed of temporally ordered data items, with no requirement of fixed periodicity. Although many of our motivating examples are media applications, streams need not be bandwidth intensive. For example, a temperature sensor that provides notification whenever the ambient temperature crosses a certain threshold is a form of streaming data. Producers place a stream's constituent data items (frames, samples, etc.) into a channel and consumers retrieve data items of interest from the channel based on timestamps and intervals. Consumers need not retrieve every item in a stream, and a common practice for consumers is the retrieval of the most current item in a stream. Buffer management/garbage collection is handled automatically.

Stampede<sup>RT</sup>'s programming model is inspired by Stampede [21], a cluster-computing middleware originally targeting a similar class of applications, although there are major and fundamental semantic differences necessitated by the nature of live streaming applications. Though Stampede<sup>RT</sup> owes its roots to the Stampede programming model, the requirements of the aforementioned applications caused it to evolve into a substantially different system (though it retains the Stampede moniker).

The two primary contributions of this paper are i) the abstract Stampede<sup>RT</sup> programming model designed to support live streaming applications (Section 3) and ii) a concrete software architecture to realize the abstractions provided by the programming model (Section 4). We also describe several relevant, motivating applications and how Stampede<sup>RT</sup> can provide convenient programming support for them (Sec-

tion 5). Finally, we provide an experimental evaluation of our initial system prototype (Section 6).

## 2. Related Work

The development of more expressive and convenient programming models to ease the burden of distributed programming is a very common goal and there is an extremely large body of prior work in this area. Nonetheless, no existing system directly provides a programming model tailored for the natural expression of idioms common in the development of live streaming applications with appropriate performance. Much work has gone into layering more convenient and structured abstractions on top of underlying unstructured transports. Remote Procedure Call [23] and Remote Method Invocation [24] are now nearly ubiquitous mechanisms to provide more structured network programming. Although very expressive and widely applicable, RPC/RMI are very general and typical implementations are unsuited for continuous bulk data transfers; the programming model of RPC makes it unnatural or impractical to exploit multicast for many kinds of interactions. Additionally, time-based manipulation of streaming data would have to be layered on top of a basic RPC system.

In many domains involving high-performance computing, message-passing systems are more common than RPC/RMI. Systems like PVM [9] and MPI [12] provide more facilities for point-to-point messaging and various collective communication operations. Although significantly more convenient than raw transport-level operations and very general, message passing systems like MPI and PVM are still fairly low-level; additionally, such systems have traditionally been narrowly targeted towards relatively static cluster-computing environments and may not handle failure or dynamism in a manner appropriate for more widely distributed environments. Various efforts have attempted to address related shortcomings: MPI-2 [18] addresses the issue of static participants by expanding the process model to allow runtime dynamism. FT-MPI [10] stands for “Fault Tolerant MPI” and attempts to address MPI’s shortcomings with regard to failure tolerance.

Configurable communication systems like Isis and Horus [26] are often targeted for group communication, but are more appropriate for applications requiring heavyweight features such as group membership agreement or causal message ordering. CCL [3] also provides a number of powerful primitives for group communication. For our target class of applications, group communication is more appropriate than point-to-point messaging, but per-stream group broadcasts would still involve much redundant messaging because each item would be broadcast to each stream consumer, even those that may not need it. Additionally, many group communication systems are not designed to support a large number of groups or groups with quickly varying membership. Finally, the recognition of time as a first-class entity would still need to be layered on top of a group-based communication system.

Although they are not traditionally used for applications with high-volume communication requirements, tuple-space programming models like Linda [4] can provide a

fairly natural mental model for stream-based processing if each stream is represented by a tuple space and a timestamp is used as the tag for each item. In order to provide automatic storage management, the runtime would need to keep track of a window of currency and reclaim items with timestamps older than the minimum bound. To the best of our knowledge, no existing tuple space implementation provides all of these features with suitable performance for real-time streaming media. TStreams [16] is a proposed model of parallel computation general enough to subsume the previous description of stream-based programming in tuple-based systems, but it is a language/compilation target rather than a middleware environment.

Some domain-specific and distributed programming languages, such as Oz [13], provide communication channels as language primitives. Efforts in streaming database research, such as Gigascope [7] or TelegraphCQ [5], deal with database-like queries over streaming data sources. These solutions typically focus on a more declarative models of stream manipulation and we see this approach as complementary to systems like Stampede<sup>RT</sup>.

Many systems exist for transport of streaming data for viewing: Yima [22] and related systems are concerned with scalable media delivery to many clients. Several systems, such as TOAST [11], MAESTRO [15] and CORBA *Audio/Video Streaming Service* [17] augment CORBA [20] to provide transport of streaming media and concentrate on efficient transmission. Nizza [25] provides a framework for the construction of real-time streaming multimedia applications, although the applications are not distributed.

Stampede [21] and D-Stampede [2] are the most closely related programming systems and are Stampede<sup>RT</sup>’s direct predecessors, but there are significant differences in the programming models, primarily in the handling of time and garbage collection. Stampede’s programming model also involved distributed data structures, but applications operate with discrete virtual time stamps, which is significantly less natural and makes synchronization difficult. Also, Stampede requires distributed upkeep of system-wide global virtual time minimum bounds in order to perform garbage collection, and channels store distributed state on behalf of clients, making transparent relocation and replication difficult. Stampede<sup>RT</sup> uses currency bounds which require only local information. Stampede also does not provide inter-stream synchronization facilities. Finally, Stampede’s programming model, like basic MPI, assumes a static participant set, which makes adapting to more widely-distributed and dynamic environments difficult. D-Stampede eliminates this limitation but at the expense of dynamic participants not being “first-class” entities (operations of dynamic participants must be proxied by a static participant).

## 3. Stampede<sup>RT</sup> Programming Model

Stampede<sup>RT</sup>’s programming model involves distributed threads of computation communicating implicitly via distributed data structures. Figure 1 shows a simple example of this paradigm. Two threads place video frames into separate channels for use by feature extractor threads. The results of the feature extraction process are placed into other chan-

nels for an analysis thread to use. The primary abstraction for stream-based data transport and manipulation is called a *channel*. Channels store discrete items sequenced by *wall clock timestamps*; channels are used to connect producers and consumers in arbitrary N-to-N configurations. Storage handling in channels is automatic and the programmer can choose from several sets of garbage collection semantics (see Section 3.4) on a per-channel basis. When using a “pull” consumption model, items of interest are fetched based on the associated timestamp information (see Section 4.1 for more on push/pull). Stampede<sup>RT</sup> also provides facilities for inter-stream synchronization based on temporal information. Time is the primary attribute used to manipulate and identify items in a channel and is discussed in detail in Subsection 3.1. Stampede<sup>RT</sup> also provides simple naming and discovery services for both peer endpoints and channels.

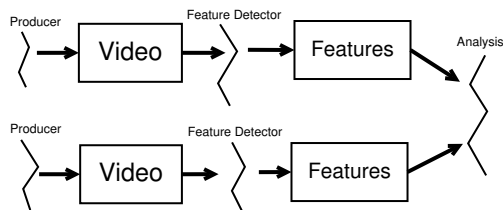


Figure 1: Conceptual Application

Fundamental aspects of the Stampede<sup>RT</sup> programming model revolve around the concept of “streams,” which we have previously defined as any continuous feed of temporally ordered data items with no requirements of fixed periodicity. Data items in a stream may also have a defined “duration.” For items in a conceptually continuous stream, such as video or audio, the duration would simply be the inverse of the production rate (33.3 milliseconds for video at 30 frames/second), because each item represents a discrete sampling of some continuous data. For other streams, the item duration may signify the length of time an item is “valid.” For the motion sensor example, an item duration may signify a time window during which no new alerts will be signaled. The concept of an item duration may not be meaningful for some aperiodic event streams.

### 3.1. “Real” Time

Channel items are indexed by wall clock timestamps, which we also refer to as “real” time (in contrast to virtual time). Although wall clock time is often a perilous issue in distributed systems, it provides the most natural mechanism for programmers dealing with streams and synchronization in live streaming applications. The alternative, virtual time, is used heavily in discrete event simulation, but can impose awkward constraints on live streaming applications. For example, media streams use the concept of real time to synchronize different components of coherent streams (e.g., separated video and audio streams corresponding to the same television program); one cannot derive the frame rate of a video stream from virtual time information alone. To work around this limitation, programmers need to keep track of various streams’ real time to virtual time correspondence explicitly. In the presence of variable-rate streams,

the programmer would need to bundle “real” timestamps explicitly along with the data payload. Since Stampede<sup>RT</sup> uses real timestamps directly, the programming model is more natural; additionally, many synchronization tasks are greatly simplified because the system recognizes real timestamps as first-class entities (as opposed to transmitting real timestamps “out of band” in the opaque data payload).

### 3.2. Channel Semantics

Each data item in a channel is associated with a timestamp specified at the time the item is placed into the channel: we call this timestamp the *production time* of the item because it is typically set to the current wall-clock time when the item is placed in the channel by a data producer. Items are ordered in a channel chronologically by production time. In this manner, each item spans a time interval from its production time until the production time of the next item (or *now* if it is the newest item available). By default, the timestamp of an item is the current time when it is placed into a channel, but explicit user-provided timestamps are also used in certain circumstances; for instance, a computation that transforms a media stream into a feature stream would likely retain the original timestamps from the media stream so there is a natural mapping between both streams. Since more than one producer may place items into the same channel, multiple items may have identical timestamps, although this is somewhat rare on platforms with high granularity timers (timestamps are currently microsecond resolution). To accommodate multiple items with the same timestamp, one can alternately view a channel as a sequence of buckets ordered by timestamp, where each bucket contains at least one item.

Items are retrieved from channels by specifying a time interval containing the items of interest. Since wall clock time is perceived as continuous, programmers will tend to work naturally with time intervals; some intervals are explicitly specified, but others may be specified using time variables (*meta-times*). For example, “5:30pm to 5:35pm” has concrete times as upper and lower bounds, while “the last thirty seconds,” is implicitly specified with the current time as an upper bound and “the current time minus thirty seconds” as the lower bound. Another common idiom is retrieving the newest item older than a certain bound (typically the last item retrieved).

Several special “meta-times” are specified for use in the construction of intervals. The special time *now* represents the current time (when a call is made) and is always the upper bound of the interval subsumed by the most recently produced item of a given channel. When a newer item is placed into the channel, the time at which it was added becomes the concrete upper bound for the previous item. Other common meta-times include *newest* and *oldest*, which are the production times (lower bounds) of the most recent and oldest items in a given channel respectively. *newest-after* is a special meta-time specifying the newest item in a channel only when the newest item is more recent than a given timestamp. Meta-times can also be offset by concrete amounts, such as “*newest* minus thirty milliseconds.”

The Stampede<sup>RT</sup> programming model does not specify

whether *push* or *pull* semantics are used for the actual transfer of remote items, but the conceptual framework specifies semantics in terms of “put” and “get” operations on items for producers and consumers, respectively. A programmer can control whether each operation is non-blocking or blocking (for instance, when a channel is full or no items are available). When a client performs a “get” operation on an interval, the system will provide all items contained fully or partially within that interval (inclusive or exclusive bounds are user-specifiable). Since Stampede<sup>RT</sup> has no application-specific knowledge of the data in channels, it cannot subdivide an item, but the application may be able to do so if such an operation is meaningful. Consequently, a “get” operation will provide whole items partially contained within the specified interval, or the user can optionally request only the items fully contained within the interval – this would be useful in the case of very large items that are also indivisible.

### 3.3. Synchronization and Channel Groups

Synchronization is an important concept for applications dealing with data streams. Frequently, video and audio corresponding to the same logical program stream will be demultiplexed into separate, atomic streams for processing (for example, separate audio and video feature extractors). It is important to provide the programmer with a mechanism to synchronize these streams. Since the process of stream synchronization is application and content-specific, Stampede<sup>RT</sup> does not explicitly synchronize content in any manner that is dependent on the type of data. It instead provides high resolution timing information and support for item-level timed synchronization of multiple channels.

In order to synchronize items across several channels, a client must perform a “get” operation from the same time interval on each channel. In many cases, a programmer may wish to get the item corresponding to *now* (or some other special time) from several channels; it is not appropriate to simply retrieve the item corresponding to *now* from different channels in sequence because new items may be produced between successive gets. Instead, the client needs to get an item corresponding to *now* in a single channel and then use the real timestamp as a reference for gets on the other channels in order to receive a coherent cross-section of items from the same time interval. The process is depicted in Figure 2. The stream that the other streams use as a synchronization reference point is called the *reference stream*. If a single thread is performing all of the gets, the programmer can perform this operation explicitly by getting an item from the reference stream and using the associated timestamp for subsequent gets; Stampede<sup>RT</sup> provides group “get” operations on multiple channels using a reference stream for this purpose. However, when consumers in separate threads need to perform the same operation on a set of channels, extra inter-thread communication would be required for synchronization. In order to alleviate such complications, Stampede<sup>RT</sup> provides system-level support for this manner of synchronization implicitly using an abstraction called *channel groups*.

A channel group is an abstraction which conceptually controls the visibility of items in a set of associated chan-

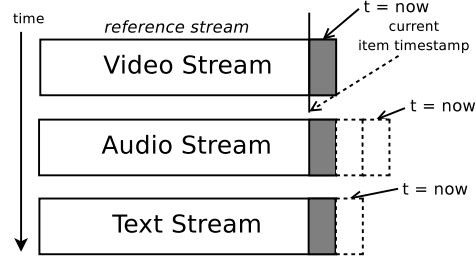


Figure 2: Reference Stream Pattern

The “now” labels depict what *now* would specify if the items were retrieved in sequence from the channels without a grouped operation. The shaded items a synchronized interval. The synchronization lower bound is the timestamp of the most current item in the reference stream.

nels without modifying the actual items available. Since the visibility information is associated with the channel group, a channel can belong to many groups simultaneously. Channel groups can be created and destroyed dynamically, and they are simply a collection of channels with a designated reference channel. The result of creating a channel group is a new set of channel descriptors: each channel descriptor refers to the original input channels, but get operations on the new descriptors will be synchronized with the reference stream. Figure 2 also depicts the result of creating a channel group on three streams. Items newer than the newest item in the reference stream are unavailable when getting from the channel group descriptors. The hidden items are still in the channel and available through the original, ungrouped channel descriptors. The programmer may also want to dynamically synchronize with the channel that lags the furthest behind at any given moment. In other words, the reference stream for the channel group cannot be set to a specific channel *a priori*, but instead must be determined by considering the newest item *i* from each channel and choosing the channel with the oldest *i*. For this purpose, Stampede<sup>RT</sup> provides a special reference stream identifier *oldest* which refers to the channel with the oldest current item.

As mentioned earlier, each item in a channel has a specific timestamp associated with it, and an item spans an interval from its timestamp (a lower bound) to the timestamp of the next item (an upper bound). Channels in any given channel group synchronize to the newest item in the reference channel for that group, making the corresponding items in the other channels visible. Since the upper bound for the newest item is not yet known, the lower bound of the newest item *l* is used for synchronization by default. All items with timestamps  $< l$  are visible, so consequently the newest item in the reference stream is hidden. The programmer may additionally provide an implicit estimate of the next item’s timestamp in the form of item durations, which provide an upper bound for the newest item. When durations are provided, the system also allows the programmer to choose to use the upper bound for synchronization where appropriate. Figure 3 shows an extended example of a channel group reference stream including item durations and hidden items. Text items are produced at the approximate rate of one item per second and have a duration of one second, while audio samples are produced at the rate of eight per second and have a duration of 125ms. If item visibility is synchronized

to the production of text items, the shaded items will be available. Several audio items newer than the upper bound of the newest text item have been produced, and they will be made visible as soon as a corresponding text item is produced. Currently, Stampede<sup>RT</sup> does not allow temporally overlapping items, so the effective duration of an item is also bounded by the production of the next item (the minimum of the specified duration and the difference in time between items). We are currently investigating the semantic implications of streams with partially overlapping items and whether they would be useful.

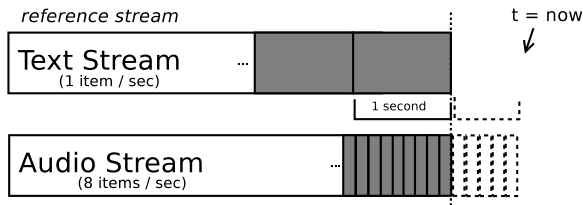


Figure 3: Item Duration Example

“now” represents the current time, and shaded items are “visible” when synchronized using the text channel as a reference stream.

Channel groups are a mechanism for coarse time-based synchronization of streams. This form of synchronization is not a direct substitute for application-specific, finer-grained/precise synchronization generally required for media playback. The primary purpose is to keep cooperating live analyses of related streams roughly synchronized.

### 3.4. Garbage Collection

Stampede<sup>RT</sup> supports several different garbage handling strategies, and the policy can be set on a per-channel basis (at channel creation time); the philosophy underlying all options leans towards simple solutions not requiring the calculation and constant upkeep of global state. One option for garbage handling is based on buffering time windows for channels: for instance, the programmer might specify that a time window of ten seconds is needed and items older than ten seconds will be discarded automatically. In a system dealing with continuous streams, an important factor is currency – a measure of how current data is. The buffering window is an implicit specification of desired currency (a lower bound) and should be a natural way to specify garbage handling for many types of media streams. The bound itself could be dynamically adjusted or calculated based on the number of consumers (and possibly other factors such as estimated end-to-end latency).

In addition to the bounded currency garbage handling model, several other simple strategies are provided for flexibility. One strategy is simply allowing a fixed number of items, where the oldest items are pushed out by newer ones. In some circumstances, applications may wish to constrain channel capacity by limiting the total data size of all items in a channel; in this case, the limit is obeyed by reclaiming the oldest items (however, as with cache-replacement policies, size-limited metrics have to be considered against the possible variance in data size between items). We are also considering the implications and utility of various hybrid poli-

cies, such as a fixed limit on items plus a lower currency bound, where a producer would block putting new data into a full channel until the oldest item passes the lower currency bound. Finally, we also provide a simple reference counting model for situations where the number of consumers is known *a priori*.

### 3.5. Other Features

Stampede<sup>RT</sup> also provides abstractions other than channels for control data or messaging between distributed components. Each client has a mailbox to receive messages from other clients. Simple distributed queues are also provided, and they can use any of the non-time based garbage collection mechanisms (in fact, a mailbox is simply a FIFO queue where each item has a reference count of one). Also provided are naming, registration and discovery services for channels and “participants” of the computation (producers or consumers of data in channels).

## 4. Architectural Elements & Implementation

This section provides a description of a concrete software architecture implementing the semantics of the Stampede<sup>RT</sup> programming model (presented in Section 3). Conceptually, the system is structured as a distributed middleware with peer-to-peer semantics – the system is designed for scalability with decentralized communication and support for features like replication and multicast. The participants and network connections are less dynamic than MANETs and pure peer-to-peer systems, but more dynamic than traditional cluster-computing models (MPI, Stampede, etc.). The nature of the target class of applications lends itself to a structure with a core of HPC resources with external leaf nodes as potential data sources. For example, a video surveillance application might use a cluster of workstations (or several federated clusters) for intensive analysis of video streams provided by many remote peers with attached cameras.

All participants have the ability to be first-class peers and some peers, currently termed “supernodes,” are responsible for storing dynamic distributed state. A supernode can be any node within the network, but the best candidates are centrally located and have higher expected connectivity. The architecture includes a registration and naming component that allows channel endpoints to be opaque. This part of the system state is distributed amongst the supernodes (forming a distributed directory of named channel endpoints). There is also a centralized front-end component for binding and location of supernodes. In principle, this component could also be distributed if it becomes a bottleneck. Stampede<sup>RT</sup>’s key architectural features are channels (and related structures, such as channel groups), peers, supernodes and the front-end component.

### 4.1. Channels and Channel Groups

Channels are the basic data abstraction in Stampede<sup>RT</sup>. Channels (and other data structures including queues, mailboxes, etc.) are hosted at a given peer, but can be replicated or migrated. The default transfer semantics for data

items are in terms of “get” and “put” operations for consumers and producers, respectively. This naturally leads to the default item transfer semantics of a “push” model for producers and “pull” model for consumers. To be more precise, there are two orthogonal dimensions of push/pull variance: the programming interface and the actual data transfer paradigm. For the programming interface, the pull model for consumers would imply explicit “get” operations, while the push model would cause a callback to be executed for each item of interest. For producers, the push model would imply an explicit “put” operation, while the pull model would use a periodic sampling of a predefined data area or periodic callbacks. The data transfer paradigm affects how remote data items are actually transferred: a push model implies that items of interest are sent when ready, while a pull model implies that items are fetched on demand.

Consumers may utilize a “push” programming interface by registering a callback to be invoked when a new item is available on a local channel. We are currently investigating the utility of a “pull” programming interface for producers in our target applications. Consumers may utilize a “push” data transfer model by registering a new item callback action on a given channel. When a new item is added to that channel, the callback will execute and send the item to the interested host or hosts. A local *proxy channel* is created on the interested host that receives items from the callback action. Local actions utilize the proxy channel as a data source. Although this strategy is simple, it is also quite versatile. It can easily support read-only replicated channels for locality by having other peers use the proxy channel or multicast by having the callback action send to a multicast group.

Channel communication can occur over various transports, and choice of transport can be negotiated by each peer/channel combination. Each peer capable of hosting resources has a “gatekeeper” endpoint using TCP/IP as a common denominator protocol. The gatekeeper is contacted for binding to channel endpoints hosted by a particular peer. After negotiation, the channel data transfer could use TCP/IP for peers linked via a WAN, a custom protocol for intra-cluster communication, or shared memory for peers collocated on the same host.

Channel groups with a concrete reference stream are implemented by simply broadcasting new item timestamps to the other streams in the group. In the case of the *oldest* meta-stream, new item timestamps are broadcast by all streams. Although this strategy is simplistic, it is typically not a bottleneck for several reasons: in most of the common anticipated use-cases for channel groups, the channels will be hosted on the same peer or peers on the same local network. This is due to the fact that channels in a group are typically closely related (e.g. part of a coherent multi-modal stream broken into components, such as video/audio or two video cameras for stereo vision). In these cases, efficient shared-memory or multicast communication can be utilized. The bandwidth for broadcasting timestamps is miniscule compared to the bandwidth required for most media streams, and the latency between hosts already provides a lower bound on the synchronization accuracy. If larger or more widely distributed channel groups are more commonly required than

anticipated, more advanced techniques can be considered.

## 4.2. Peers

A peer in Stampede<sup>RT</sup> is simply a single distinguished participant in a distributed application. Stampede<sup>RT</sup> does not dictate any particular mapping between peers and threads or processes. Peers are identified by opaque unique identifiers and can also be assigned unique names. Peer names are useful in creating channel endpoints hosted by a particular peer (they are also used for mailboxes). Peers can join and depart the communication dynamically, although peers hosting resources may need to negotiate the migration or shutting down of data sources in use.

Each peer has several local caches to store mappings between opaque host/channel identifiers and transport endpoints. In the common case, an operation on a channel (getting or putting an item, for example) will simply require a table lookup for a cached endpoint descriptor. If the cache lookup fails or the endpoint is found to be invalid, the system will attempt to resolve the missing connection endpoint by contacting an appropriate entity. The major cached entities are *host gatekeeper endpoints*, *channel data endpoints* and *channel identifier to host identifier mappings*. A cache miss on a channel data endpoint will cause the system to contact the host gatekeeper for channel information. A cache miss on the host gatekeeper endpoint table or the channel identifier to host identifier mapping will cause the system to contact a supernode for appropriate information. It is also possible the cached mappings are stale; in this case, the operation will fail and will be treated as a cache miss. The form of cached channel data endpoints varies depending on the transport, and the table containing channel data endpoints is indexed by the pair consisting of the opaque channel identifier and the local thread identifier (if applicable); the inclusion of the thread identifier allows multiple threads in the same peer to have individual persistent connections to the same channel.

## 4.3. Supernodes

Supernodes form a loosely consistent distributed directory. Our definition of a “supernode” is similar to that in several P2P systems like FastTrack and Gnutella [6]. Any peer in Stampede<sup>RT</sup> can “volunteer” to be a supernode, but the intended system configuration has a small number of supernodes hosted by peers with high availability and connectivity. Peers will preferentially order their list of supernodes to contact by latency, so it is beneficial to place supernodes within various network segments to provide low-latency operation to local peers.

Supernodes store several directories of interrelated information: 1) a directory mapping *opaque host identifiers to gatekeeper endpoints*, 2) a directory mapping *opaque channel identifiers to host identifiers*, 3) a directory mapping *channel names to opaque channel identifiers*; and 4) a directory mapping *host names to opaque host identifiers*. Peer queries to supernodes are straightforward operations, and updates cause propagation to all other supernodes. Supernodes are kept synchronized using standard consensus techniques. Although the updates require agreement among the

supernodes and therefore the cost grows with the number of supernodes, directory update operations are relatively infrequent and are not “critical path” operations.

Like peers, supernodes can join and leave the system at will. However, since supernodes are typically hosted on high-availability nodes and use minimal system resources, we expect that supernodes will rarely leave the system voluntarily (since supernodes are “volunteers,” there is no election process/handoff mechanism if no supernodes remain).

#### 4.4. Prototype Implementation

The major features of the proposed programming model have been implemented. This set of features includes “real time” semantics, channels, time interval get operations, channel groups, and channel callbacks (to implement proxy channels, for instance), plus the described peer architecture. The prototype is implemented in plain ANSI/ISO C89 with the additional requirement of some common basic library functionality (POSIX threads, writev, gettimeofday, XDR routines, etc.). The current prototype does not implement synchronization between supernodes for registration/naming information, so ongoing work includes the full implementation of synchronization between supernodes and the addition of shared-memory and other transport choices besides TCP for data transfer between peers.

### 5. Applications

This section describes three concrete, motivating applications that Stampede<sup>RT</sup> is designed to support. These applications also exhibit a range of diverse requirements for live streaming applications. TV Watcher is a distributed application requiring user-directed intensive media analysis on HPC resources with additional lower-end viewing clients. MediaBroker [19] is a higher-level middleware for constructing a certain class of pervasive distributed applications. Finally, LAGR is a distributed robotics application with soft real-time requirements running on a small multi-computer robot.

**TV Watcher:** TV Watcher [14] is an application designed to help a television viewer deal with “information overload” associated with a large number of television channels. It allows a user to navigate through multiple televised video streams and identify currently relevant content by selecting a live television stream representative of their current interests and correlating it with other available television streams. The information used for correlation can also be used to locate relevant webpages via a web search plug-in.

The implementation of TV Watcher utilizes D-Stampede [2] for stream transport underneath a higher-level middleware called *Symphony*. Many of the features in *Symphony* are designed to work around limitations of D-Stampede’s programming model when dealing with a variety of media streams. Ultimately, Stampede<sup>RT</sup> would almost entirely obsolete *Symphony*, obviating the need for much of the naming, registration and discovery components, as well as the ad-hoc layering of wall-clock time at the layer above Stampede. The recognition of wall-clock time at the transport abstraction level also increases efficiency, because items

need not be unnecessarily retrieved from remote locations to check their wall-clock timestamp. Stampede<sup>RT</sup>’s channel groups would also provide a direct idiom for synchronizing the audio, video and closed-captioning text streams corresponding to a single television program. Although TV Watcher is no longer under development, an implementation of TV Watcher using Stampede<sup>RT</sup> would require relatively few changes to the application and completely replace most of the lower-level time and stream-based functionality of *Symphony*.

**MediaBroker:** MediaBroker [19] and its successor MediaBroker++ are middleware systems intended to support a class of pervasive applications needing access to both HPC resources and a wide variety of end-devices embedded in the environment. MediaBroker is a higher-level middleware than Stampede<sup>RT</sup> and manages application-level computation to some extent. MediaBroker itself requires a stream-oriented transport layer that is provided by a lower-level middleware. MediaBroker’s current implementation could be greatly simplified by using Stampede<sup>RT</sup>. In fact, our experience with developing MediaBroker and its successor partially motivated the development of Stampede<sup>RT</sup>.

MediaBroker provides a framework for automatically instantiating computations related to type transformations in applications specified by coarse-grained dataflow graphs. It facilitates the interaction of a wide variety of end-devices utilizing heterogeneous data formats and provides a mechanism for automatically handling application-level computation. All data streams are typed and computation is described in terms of type transformation.

The current iteration of MediaBroker is partially transport agnostic in that it does not define a particular programming model for type transformation code. The developers of transformation code will derive the most benefit from more natural transport abstractions. Clients may be able to make use of the synchronization facilities appropriately exposed through the MediaBroker interface. In addition, the runtime system will be simplified with Stampede<sup>RT</sup>’s straightforward garbage-handling facilities. Ultimately, MediaBroker’s flexibility also depends upon the underlying transport allowing a level of dynamism and a mix of first-class participants across devices with significantly different levels of resources and expected availability.

**LAGR:** LAGR (Learning Applied to Ground Robots) is a DARPA program [8] with the goal of developing high quality control software for autonomous ground robots. A number of universities and research labs have developed distributed robotics application designed to run on multi-computer robots. One of the motivating applications for Stampede<sup>RT</sup> is the Georgia Institute of Technology’s LAGR software system, which we will simply refer to as LAGR. The current platform for LAGR utilizes a custom, message-based communications library.

An effort to utilize Stampede<sup>RT</sup> in LAGR is currently underway, and LAGR’s developers provided feedback during the development of Stampede<sup>RT</sup>’s requirements. The system performs stereo reconstruction on video frames from two camera views, so each camera will have a video chan-

nel and a channel group will be used to synchronize the two. Communication between the planning, mapping and stereo vision modules will use channels and mailboxes. The move from point-to-point message-oriented communications to distributed shared data structures will also make the addition of extra analysis modules simpler (because adding extra consumers requires no changes on the part of producers).

The initial changes required in LAGR are minimal since the code does not use the communications library directly; instead its use has been isolated behind an API already similar to the interface that Stampede<sup>RT</sup> provides. Initially, the move only required a change of several lines of code in each application (to initialize the library). Additionally, many objects include real-time timestamps explicitly in their serialization and this is now unnecessary. However, some of the code could be slightly restructured in order to better take advantage of Stampede<sup>RT</sup>'s facilities. All of these changes taken together will reduce the lines of code in LAGR required for communications-related functions.

## 6. Preliminary Performance

In this section we present a series of microbenchmarks and experimental results designed to highlight the features of the programming model and evaluate the performance of key primitives of Stampede<sup>RT</sup>. We present three sets of experiments, the first of which is a set of micro-benchmarks designed to measure the local cost of channel primitives. Next we present a series of benchmarks to demonstrate that channels can scale well when many consumers are contending for access. Finally, we demonstrate that channel groups perform as expected in a realistic application scenario.

**Channel Primitive Micro-benchmarks:** Figure 6 depicts a set of micro-benchmarks assessing the cost of local channel operations. These microbenchmarks are run on a 2.2Ghz dual-core Opteron workstation with 2GB of RAM running 64-bit Linux 2.6.17.1 with a preemptable kernel. We measure the cost of a local item retrieval operation because it is more complex than placing an item in a channel and requires a traversal of the same data structures. Figure 6 shows the cost of retrieving one item by timestamp from a channel while increasing the number of items (by an order of magnitude each time – the x-axis scale is logarithmic). Each item is 64 bytes, although the size of the item does not affect the retrieval time for a local operation since item data is returned by reference. Each result is averaged over five runs and the cost of a single operation is derived from a measurement of 10,000,000 identical operations performed sequentially. Due to the internal data structures used for bookkeeping, the cost of retrieving an item from a channel will vary between different items; consequently, the cost is measured for three different timestamps (one near the beginning, middle and end in the temporal sense). Both graphs show the worst-case observed<sup>1</sup> and average values. When increasing

<sup>1</sup>Measuring the absolute theoretical worst case is complicated due to the use of several different data structures with amortized access times. However, a brute-force test for smaller sizes (1001 and 10001) revealed that the absolute worst-case behavior did not deviate more than a few percent from those presented.

the number of items in an interval, the get time increases approximately linearly with the number of items<sup>2</sup>.

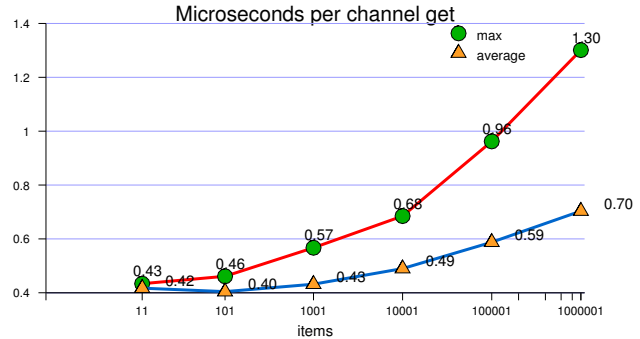


Figure 4: Channel Get Micro-benchmark

**Channel Scalability Benchmarks:** The remainder of the benchmarks are performed on a cluster of dual-processor 3.06 GHz Intel Xeon nodes with hyperthreading enabled and 1 GB of memory. Each node runs 32-bit Linux 2.6.9 and the nodes are interconnected with Gigabit Ethernet. The channel scalability benchmark tests the ability of a single channel to serve many consumers with minimal frame loss. The producer puts frames of a Motion JPEG video stream into a channel at approximately 30 frames per second, where each frame averages about 18KB and the total number of frames is 2312 (thus each run lasts approximately 1 minute 15 seconds). Each consumer retrieves the newest frame repeatedly, blocking if the last frame read is the newest frame available. Each consumer negotiates a persistent, dedicated TCP connection to the peer hosting the channel (in this case, the producer).

As we scale the number of consumers contending for access to the same channel, we measure the number of dropped frames by consumers. Each dual-processor node runs a maximum of four consumers and the producer runs on a separate node. Each experiment configuration is run five times for both Motion JPEG video and uncompressed RGB video, and Figure 5 shows two metrics: the sum of all frames dropped between all consumers (averaged over the five runs) and the maximum number of frame drops by any single consumer on any run. Even with 32 simultaneous consumers, the maximum number of frames dropped by any client on any run of the Motion JPEG test was four, and less than seven frames were dropped on average between all 32 consumers. A similar test was repeated for a frame size of 300KB (a frame size appropriate for uncompressed RGB video frames). With a significantly higher data rate, the single unreplicated channel cannot serve as many consumers, but still scales quite well, dropping only 9.5 frames on average between all 12 consumers. With 16 consumers (not shown in the graph), the load is simply too high and the average consumer drops approximately 17% of frames, with the maximum loss rate for a single consumer at ~24% (dropping 549 out of 2312 frames).

<sup>2</sup>Due to space constraints, this graph is not included. A tech report version will contain elided graphs.

The theoretical data rate for 16 consumers is greater than a single gigabit interface could support, but we can use replicated *proxy channels* (see Section 4.1) to split the load between two hosts. Figure 6 shows the results of using a single proxy channel with the uncompressed video feed. The experiment setup is essentially the same as the previous experiment (Figure 5) with the addition of the proxy channel running on a separate host: the first 11 clients are served by the original producer, with the remaining clients assigned to the proxy. These results demonstrate that channels can scale well to serve multiple consumers, even with a relatively high data rate.

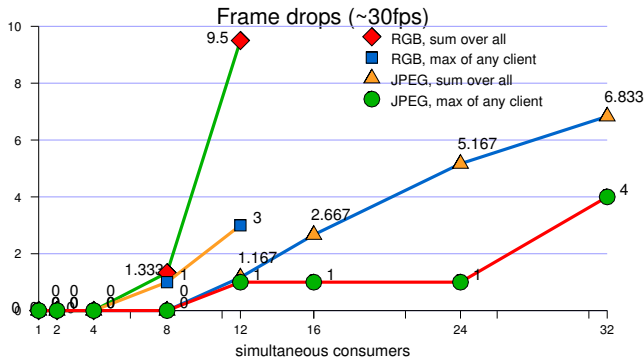


Figure 5: Channel Scaling Benchmarks

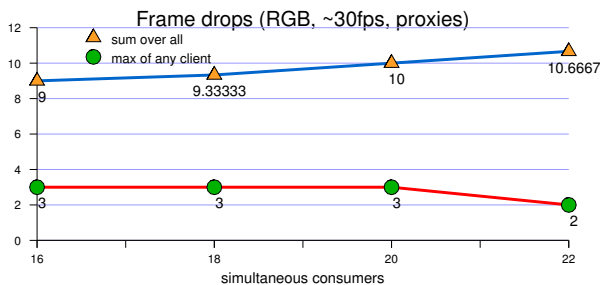


Figure 6: Channel Scaling with Replication (proxies)

**Channel Group Benchmarks:** In order to assess the properties of channel groups, we utilize a kernel of a real application (extracted from the *TV Watcher*). This example uses two producer threads on the same peer, one producing motion JPEG video frames at 30 frames per second, the other producing decoded closed captioning text at the rate of two items per second (each item is 0.5KB). A consumer on one machine retrieves video frames and a consumer on another machine retrieves closed captioning text items. In one case, the consumers both retrieve from channel descriptors that are part of a channel group (synchronized dynamically with the meta-stream *oldest*), while in the other case they simply retrieve new items as they are available. All three peers (the producer and two consumers) run on different machines. Figure 7 measures the average skew per frame in milliseconds. The total skew is calculated by taking the square root of the sum of the square of the differences between retrieval times of items from the two con-

sumers<sup>3</sup>. Clearly channel groups lead to significantly lower skew and, although these results are somewhat obvious, the measurements simply provide a compact characterization of the performance of channel groups and show channel groups do provide viable synchronization in a realistic application scenario. While it is also true that the difference in skew between the two configurations can be made arbitrarily large by simply tweaking the relative rates of the two consumers, this set of parameters is taken from a real application. With channel groups, the visibility behavior of items in a channel operates like the diagram in Figure 3 (in Section 3). Since Stampede<sup>RT</sup> supports interval get operations, the video consumer with channel groups gets fifteen frames in a single operation when each new text item becomes available.

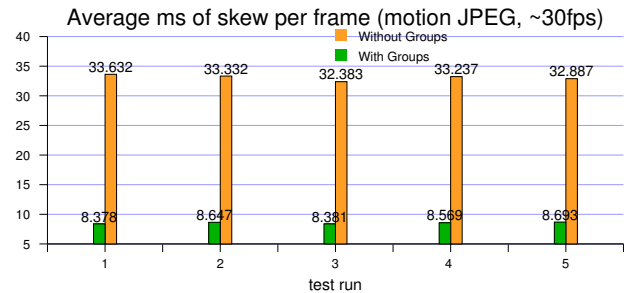


Figure 7: Channel Group Benchmarks

The previous experiments demonstrate that the key primitives in the programming model (channel operations) do not impose significant overhead, even with a large number of potential items in each channel. Additionally, contention does not significantly impede the scalability of channels to serve many consumers, even with high data rates. Finally, channel groups operate as expected, reducing skew between get times of consumers on different hosts for items corresponding to the same temporal intervals, allowing consumers to operate on related streams in lockstep without explicit synchronization.

## 7. Conclusion

Applications requiring analysis and correlation of multiple live streaming data sources are becoming increasingly pervasive. Our experiences with several such applications and higher-level middleware systems targeted towards such applications have ultimately prompted the development of Stampede<sup>RT</sup>. Although traditional cluster programming systems in the HPC domain are powerful and fast, the abstractions are often relatively low-level. In addition, many HPC-oriented systems are designed for a cluster-centric application model. Stampede<sup>RT</sup>'s goals are to provide more natural abstractions for live streaming applications while retaining good performance, with the architecture supporting a higher level of dynamism than typical cluster-oriented systems. We have presented the Stampede<sup>RT</sup> abstract programming model, featuring channels for streaming data transport.

<sup>3</sup>Each text item demarcates an “epoch” spanning approximately 15 video frames (until the next text item), and the differences are measured between video frames and text items of matching epochs.

These abstractions allow the manipulation of items and synchronization directly via specification of wall-clock time intervals. We have also described a software architecture to realize the abstractions provided by the Stampede<sup>RT</sup> programming model while maintaining good performance and scalability. The architecture has peer-to-peer aspects in order to provide a level of dynamism, allowing data providers to negotiate a data transport mechanism on a per-client basis, providing support for proxy channels, and allowing channels to move between hosts. Finally, we have described several motivating applications and presented an experimental evaluation of our Stampede<sup>RT</sup> prototype.

Ongoing work includes the use of Stampede<sup>RT</sup> in LAGR and MediaBroker/MediaBroker++, and the completion of the fully distributed registration/naming architectural component. There are many avenues of future exploration related to Stampede<sup>RT</sup> (and live streaming applications in general) including QoS considerations, fault tolerance and support for mobile devices with power constraints. Another important avenue of research, explored by systems like Crest [1], is developing support for persistence of data in such applications while still allowing queries based on temporal properties, as well as the development of abstractions allowing uniform treatment of both live streaming and archived data. Additionally, future novel applications will undoubtedly evince the need for even richer temporal semantics, additional abstractions or architectural enhancements, which we will explore as our experience with Stampede<sup>RT</sup> grows.

## Acknowledgements

The work has been funded in part by an NSF ITR grant CCR-01-21638, NSF NMI grant CCR-03-30639, NSF CPA grant CCR-05-41079, and the Georgia Tech Broadband Institute. The equipment used in the experimental studies is funded in part by an NSF Research Infrastructure award EIA-99-72872, and Intel Corp. We would like to thank Jim Rehg, Hasnain Mandviwala, Kathleen Knobe, Dave Lilithun and our anonymous reviewers for their feedback and insight regarding this ongoing work. Finally, we would like to thank members of the Embedded Pervasive Lab (EPL), College of Computing, and Georgia Tech.

## References

- [1] S. Adhikari. *Programming Idioms and Runtime Mechanisms for Distributed Pervasive Computing*. PhD thesis, College of Computing, Georgia Institute of Technology, 2004.
- [2] S. Adhikari, A. Paul, and U. Ramachandran. D-Stampede: Distributed Programming System for Ubiquitous Computing. In *Proceedings of ICDCS '02*, pages 209–216, July 2002.
- [3] V. Bala et al. CCL: A Portable and Tunable Collective Communication Library for Scalable Parallel Computers. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):154–164, 1995.
- [4] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.
- [5] S. Chandrasekaran et al. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proceedings of CIDR '03*, January 2003.
- [6] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. Making Gnutella-like P2P Systems Scalable. In *Proceedings of ACM SIGCOMM '03*, August 2003.
- [7] C. Cranor et al. Gigascope: A Stream Database for Network Applications. In *Proceedings of SIGMOD '03*, pages 647–651, New York, NY, USA, 2003. ACM Press.
- [8] DARPA: Information Processing Technology Office. Learning Applied to Ground Robots (LAGR). <http://www.darpa.mil/IPTO/programs/lagr/index.htm>.
- [9] J. Dongarra, G. A. Geist, R. Manchek, and V. S. Sunderam. Integrated PVM framework supports heterogeneous network computing. *Computers in physics*, 7(2):166–174, – 1993.
- [10] G. E. Fagg and J. J. Dongarra. FT-MPI: Fault Tolerant MPI, supporting dynamic applications in a dynamic world. *LNCS*, 1908:346–353, 2000.
- [11] T. Fitzpatrick et al. Design and Application of TOAST: An Adaptive Distributed Multimedia Middleware Platform. In *Proceedings of IDMS '01*, pages 111–123, London, UK, 2001. Springer-Verlag.
- [12] M. P. I. Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, 1994.
- [13] S. Haridi et al. Programming languages for distributed applications. *New Generation Computing*, 16(3):223–261, 1998.
- [14] D. Hilley et al. TV Watcher: Distributed Media Analysis and Correlation. CERCS Tech Report GIT-CERCS-04-25, Georgia Institute of Technology, 2004.
- [15] J. W.-K. Hong et al. Design and implementation of a distributed multimedia collaborative environment. *Cluster Computing*, 2(1):45–59, 1999.
- [16] K. Knobe and C. D. Offner. TStreams: How to Write a Parallel Program. Technical Report HPL-2004-193, HP Laboratories Cambridge, October 2004.
- [17] E. Lamboray, A. Zollinger, O. G. Staadt, and M. Gross. Interactive multimedia streams in distributed applications. *Computers & Graphics*, 27:735–745, 2003.
- [18] Message Passing Interface Forum MPIF. MPI-2: Extensions to the Message-Passing Interface. Technical report, University of Tennessee, Knoxville, 1996.
- [19] M. Modahl et al. MediaBroker: An Architecture for Pervasive Computing. In *Proceedings of IEEE PerCom '04*, Orlando, FL, March 2004.
- [20] Object Management Group. The Common Object Request Broker: Architecture and Specification, October 2000. Revision 2.4.
- [21] U. Ramachandran et al. Stampede: A Cluster Programming Middleware for Interactive Stream-Oriented Applications. *IEEE Transactions on Parallel and Distributed Systems*, 14(11):1140–1154, 2003.
- [22] C. Shahabi, R. Zimmermann, K. Fu, and S.-Y. D. Yao. Yima: A second-generation continuous media server. *Computer*, 35(6):56–64, 2002.
- [23] R. Srinivasan. RPC: Remote Procedure Call Protocol Specification Version 2. RFC 1831 (Proposed Standard), Aug. 1995.
- [24] Sun Microsystems. Java RMI. <http://www.java.sun.com/>.
- [25] D. Tanguay, D. Gelb, and H. H. Baker. Nizza: A Framework for Developing Real-time Streaming Multimedia. Technical Report HPL-2004-132, Mobile and Media Systems Laboratory, HP Laboratories Palo Alto, August 2004.
- [26] R. Van Renesse, T. M. Hickey, and K. P. Birman. Design and Performance of Horus: A Lightweight Group Communications System. Technical Report TR94-1442, Cornell University, Ithaca, NY, USA, 1994.