MediaBroker: A Pervasive Computing Infrastructure for Adaptive Transformation and Sharing of Stream Data

Umakishore Ramachandran Martin Modahl Ilya Bagrak
Matthew Wolenetz David Lillethun Bin Liu James Kim
Phillip Hutto Ramesh Jain

College of Computing, Georgia Institute of Technology
801 Atlantic Drive, NW, Atlanta, GA 30332-0280, USA

Abstract

MediaBroker is a distributed framework designed to support pervasive computing applications. Key contributions of MediaBroker are efficient and scalable data transport, data stream registration and discovery, an extensible system for data type description, and type-aware data transport that is capable of dynamically transforming data en route from source to sinks. Specifically, the architecture consists of a transport engine and peripheral clients and addresses issues in scalability, data sharing, data transformation and platform heterogeneity. Details of the MediaBroker architecture, implementation, and a concrete application example are presented in this article. Experimental study shows reasonable performance for selected streaming media-intensive applications. For example, relative to baseline TCP performance, MediaBroker incurs under 11% latency overhead and achieves roughly 80% of the TCP throughput when streaming items larger than 100 KB across our infrastructure. The EventWeb application demonstrates the utility and graceful scaling of MediaBroker for supporting pervasive computing applications.

Key words: Pervasive computing, Sensor-based distributed computing, Streaming data, Dynamic adaptive systems

Email addresses: `rama@cc.gatech.edu` (Umakishore Ramachandran), `mmodahl@gmail.com` (Martin Modahl), `reason@cc.gatech.edu` (Ilya Bagrak), `wolenetz@cc.gatech.edu` (Matthew Wolenetz), `davel@cc.gatech.edu` (David Lillethun), `bliu@ece.gatech.edu` (Bin Liu), `james.earl.kim@gmail.com` (James Kim), `pwh@cc.gatech.edu` (Phillip Hutto), `jain@ece.gatech.edu` (Ramesh Jain).

1 Introduction

Recent proliferation of special-purpose computing devices such as sensors, embedded controllers, handhelds, wearables, smart phones and power-constrained laptops marks a departure from established tradition of general-purpose desktop computing. Disparity in computation and communication capabilities between the smallest network-aware device and high-performance cluster machine or grid on the other end of the hardware continuum is rapidly increasing. The vision of pervasive computing suggests inclusion of devices spanning the entire hardware continuum. One major challenge with developing efficient pervasive applications is finding the right framework to ease their construction. In search of a solution, we investigate concrete applications currently being researched in the smart space [1] domain.

In pervasive computing environments, there is a need to efficiently transport streaming data between sources (data producers) and multiple sinks (data consumers), while those sources and sinks may have competing data type requirements. The goal of MediaBroker is to solve the problems of efficient data transport and sinks that require different data types. That is, different sinks may wish to consume the same information from a source but require that it be presented to them in different manners.

An example of an application in the smart space domain that requires both efficient data transport between sources and sinks, as well as the provision of multiple data types is *EventWeb*. EventWeb allows live event inferencing from distributed capture of raw media streams. It consists of myriad live event feeds that are queried by the user [2]. With disparate applications in EventWeb accessing similar feeds (raw and event), there should be a facility to share these resources. A camera used to monitor a room for a security system could be used by another application to record meetings in that room, for example.

EventWeb requires acquisition, processing, synthesis, and correlation of streaming high bandwidth data such as audio and video. Although each pervasive computing application like EventWeb may have distinct requirements, the common goal of bridging heterogeneous platforms to provide uniform and predictable means of data distribution warrants infrastructure-level support. Consequently, this infrastructure must accommodate, without loss of generality, all device types regardless of breadth of purpose and provide means, in terms of functionality and corresponding APIs, to develop a variety of distributed applications.

In this article, we present our contributions: (1) MediaBroker architecture design addressing data typing, transformation and data sharing requirements, (2) prototype MediaBroker implementation, (3) performance analysis of our implementation, and (4) qualitative demonstration and quantitative analysis of a concrete application, EventWeb, enabled by MediaBroker.

The rest of the article is organized as follows. We discuss requirements of target applications in Section 2. We present MediaBroker architecture in Section 3 and implementation in Section 4, followed by performance analysis of MediaBroker implementation in Section 5. Section 6 presents a demonstration of MediaBroker utility and performance in the context of supporting a concrete EventWeb application. We present a survey of related work in Section 7 before concluding our findings in Section 8.

2 Requirements

In order to manage streams in a pervasive computing environment, MediaBroker must meet these core requirements: (1) efficient and scalable data transport from sources to multiple sinks, (2) dynamic registration and discovery of data streams, (3) ability to specifiy and request data types, and (4) data transformation to fulfill the data type requests.

Requirements for the MediaBroker architecture also follow from the environment in which it operates and from applications such as EventWeb. Highly connected smart space applications encompass multitudes of sensors and demand scalability from the underlying infrastructure. High bandwidth media streams integral to these applications require system support for high throughput, low latency transport and a convenient stream abstraction. Dynamism involved in allowing sensors and actuators to join and leave the system at runtime demands a high degree of adaptability from the underlying infrastructure. The resulting additional requirements are (5) scalability, (6) low latency and high throughput, and (7) adaptability.

The scalability requirement is perhaps the most important requirement for any distributed application and is especially true for our application space, where a large number of devices may simultaneously make use of the underlying infrastructure on behalf of one or more concurrent applications. The overall expectation for a viable distributed runtime is not to give up performance to accommodate an increase in the number of active devices. Throughput becomes a bottleneck if data is produced faster than it can be transferred, and latency determines currency of data, critical in pervasive computing applications.

Adaptability entails several distinct requirements. First, it assumes devices can establish new and destroy existing communication channels at runtime while maintaining system consistency and stability. For instance, a user may choose to include a new camera in an EventWeb query at any point in time. The system has to continue uninterrupted through these disturbances. Adaptability also refers to the architecture's ability to "shift gears" per a consuming party's request. A consuming client may request to share a data stream with another consuming party but would like to receive data of a lower fidelity. The architecture must be adaptable enough to accommodate runtime changes of media sink requests.

2.1 API Requirements

Given the diversity of devices in our application space, we observe that a facility for transforming data en route is important. Therefore, major requirements to support such transformations are (1) a common type language for describing transported data, (2) a meaningful organization of transform functions applied to data, (3) a flexible stream abstraction for data transfer that allows data sharing and (4) a uniform API spanning the entire hardware continuum.

Consider applications where available sensors are embedded controllers with attached camera devices. The desired outcome is that our system is able to utilize all camera sensors regardless of feature set, but each camera model may produce video in different formats, resolutions, frame rates, or color spaces. The inherent diversity of devices and data streams calls for a standard, yet highly flexible facility for composing data type descriptions. Thus, an important requirement for any proposed infrastructure is a language enabling definition and exchange of type properties. Selection of a standard way to describe data types eases implementation of data transformation logic as well. Type descriptions not only provide necessary context for when a transform from one type to another is meaningful, but they can also serve as unique tags by which an implementation can archive and retrieve type transform functions.

The prevalent pattern of data production within our application space is a stream. Be it a camera device, a microphone, or a primitive sensor; each can be viewed as producing data in streams. A video or audio stream is meaningful to the consumer as long as data is received in exactly the same order it is produced and in a timely manner. Thus, any proposed architecture must support a notion of stream and offer a stream-like API to any application that relies on data streaming for its functionality.

Finally, a successful solution will undoubtedly include an API portable to many different platforms and transports. Portability will ensure that the distributed system can be utilized from a maximum number of devices.

3 MediaBroker Architecture

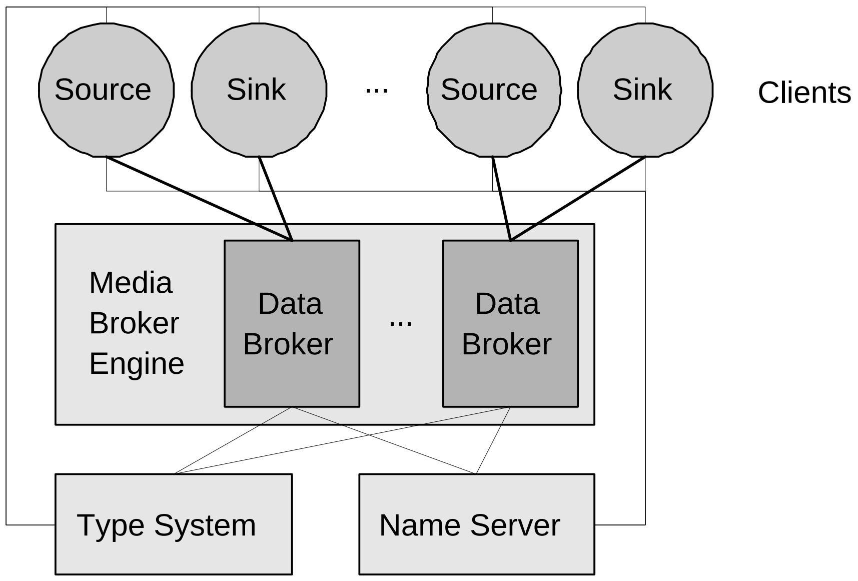

Fig. 1. MediaBroker Architecture with Example Clients

The principal components of MediaBroker (MB) architecture are the clients, transport and transformation engine, type system facility, and name server, shown in figure 1. The type system and name server form the foundation on which higher level components depend. We employ the term *client* to describe an entity that acts as a media source (data producing), media sink (data consuming) or both.

Figure 2 demonstrates basic data flow between clients and the transport engine. By making the client API available within the engine, we enable *colocated* clients to share the computational resources of the engine. In the diagram, P_1 produces media while C_2 and the colocated client consume its stream. The colocated client takes data from P_1, modifies it and redistributes it to C_3. Negotiation of data types by MB results in the engine requesting a type from P_1 that is compatible with both C_2 and the colocated client's request.

The core of the MB engine is a collection of data brokers. A data broker is an execution context within the MB engine that satisfies the data sharing requirement. It transports data from one media source to one or more media sinks. A data broker is also responsible for type negotiation between the media source and media sinks attached to it. For instance, it must address the issue of multiple media sinks requesting data from the same media source in multiple formats. It is important to note that several data brokers and clients can be "stacked" on top of one another, as demonstrated in Figure 2.

The name server, shown in figure 1, is a crucial component of MediaBroker

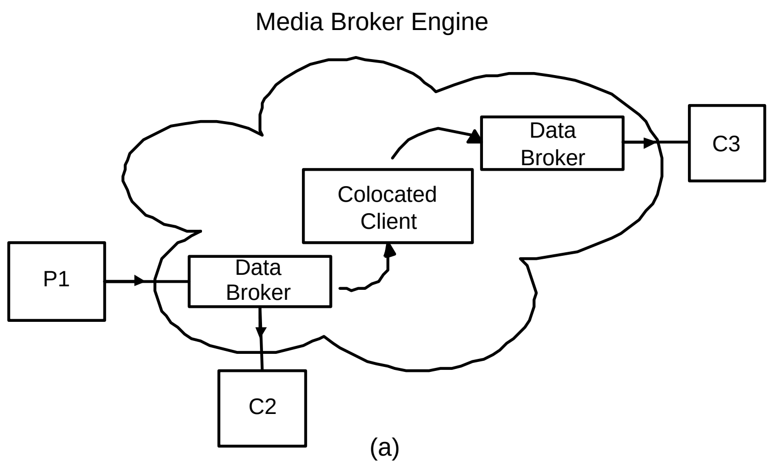

Fig. 2. MediaBroker Engine Overview

because it maintains a mapping between producer and consumer names and computation resources and data structures associated with them at runtime. Since many components depend on reliable name server information, write access to a name server is restricted to the MB engine, while external clients may read only.

3.1 MediaBroker Clients

Since multiple sinks are allowed within a single client instance, the client can perform simple data aggregation. In fact, the MB client API permits a client to instantiate or terminate any number of media sinks or media sources. As seen in Figure 2, a single colocated client is both a media sink and a media source as it consumes data from P_1 and produces data for C_3. When a new source is instantiated by a client, MB allocates a new data broker and binds it to the newly instantiated media source. As each source relies on a dedicated data broker, the system will scale as long as computing and communication facilities exist for each new data broker/media source pair. A new sink instantiated by a client is bound to an existing data broker.

An important piece of the engine's functionality is allowing new clients to join and leave at runtime. A new client begins by connecting to the MB engine, at which point the engine establishes a command channel to the client and associates a listener entity with that channel. Whenever a client decides to introduce a new media source, it signals the listener accordingly and provides non-volatile attributes describing the source, such as its name. The listener creates a data broker for the connecting media source, creates a record in the name server describing the new source, notifies the client of success, and gives it the data broker's handle. Likewise, when a client creates a new sink, it transfers the sink's attributes to the listener. The fundamental distinction is that media sinks also provide the name of the media source from which they want to consume. The listener queries the name server to find the data broker that corresponds to the named source. On success, it notifies the data broker of the media sink addition and gives the data broker's handle to the sink.

3.2 Data Broker

A high-level view of a data broker is presented in Figure 3(a). In order to satisfy both the data sharing and data transformation requirements, the data broker monitors the status of the attached media source and commands from all of its attached media sinks. Both source status and sink commands are formatted as a data type, which will be detailed in the next section. Status signifies the provided data type, while commands signify the data type requested. The data broker then reconciles the source's status with the sinks' commands by finding the least upper bound (LUB) between the types of data requested by media sinks. Once the LUB type is found, the data broker communicates it to the producer through the command channel, expecting it to produce the same. The data broker then performs the necessary data transformations from the LUB type to each of the types demanded by data sinks.

There are two scenarios in which runtime adaptability of the system occurs. One is prompted by a media sink sending a command that it would like to consume data of some new type. Following a new media sink command, output to the media sink is paused while the data broker calculates the new LUB type and determines appropriate transformations that need to be applied based on the new LUB type. Another scenario involves a media source changing its status type which has the effect of pausing *all* sinks while the LUB type is recalculated along with the transformations that need to be applied to satisfy *every* sink's request. Both scenarios imply that one or more media sinks must be paused while the data broker effects the required adaptation. Quantitative measure of adaptation latency is presented in Section 5.4

3.3 Type System

A language for describing data types and their relations to each other defines a *type* as a tuple of type domain and a set of type attributes. *Type domain* is a high level description of type, e.g. PCM audio. *Type attribute* consists of an attribute name-value pair. *Attribute name* is an ASCII string, and *attribute value* is the value of that attribute. Figure 3(b) shows several sample types.

A *type map*, bearing strong resemblance to type lattices [3], is our data struc-

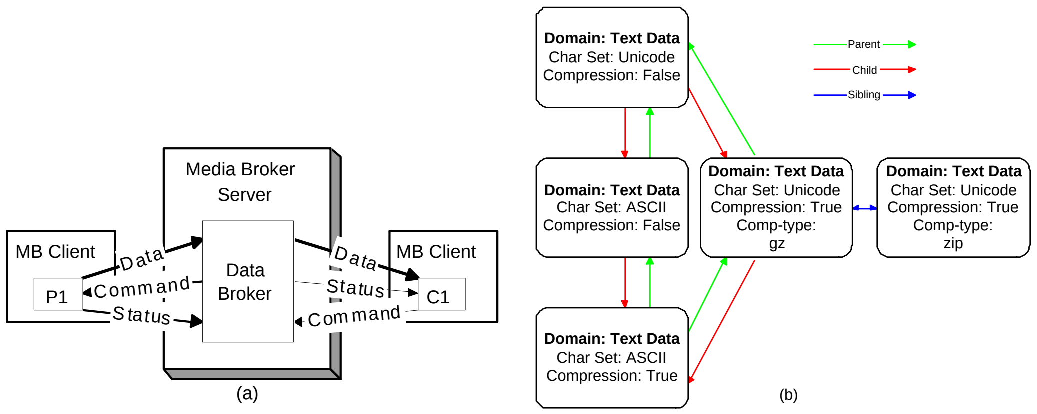

Fig. 3. Data Broker with Source and Sink (a) and Sample Type Map (b)

ture for describing relationships between various data types. Types are connected to each other via directed edges. A type map consists of a non-empty set of types, two designated root types, and a collection of directed edges. We define three types of edges: (1) *parent*, (2) *child*, and (3) *sibling*. Root types are defined as those that either have no parent or child edges. Thus, one root type has no parent edges (highest type) and the other root type has no child edges (lowest type). A *path* is defined as a sequence of edges. A well-formed type map has a path between any two types. Figure 3(b) shows a simple type map consisting of five data types.

Normally, type map information is interpreted as follows. The "highest type" is the best quality type of data that a producer is capable of supplying. Likewise, the "lowest type" is the worst quality type of data. Quality is left intentionally vague. For instance, applications may choose to equate lowest quality to be least resource consuming while the highest quality to be exactly opposite. Figure 3(b) supports this notion of quality as well. Edges connecting types signify that data can be converted from one type to another.

A type map also contains optional type transformation functions associated with the edges of the type map. These function registrations are unidirectional, so separate functions could be registered for transforming from a parent to a child and for transforming from that child to the same parent, for example. Data Brokers traverse type map edges to assemble a chain of transformation functions that transform the data type provided by a source to the data type requested by a client. There is no requirement that every edge have a transformation function registered, but as a practical matter there should be at least one chain from any node to any lower node in the type map.

Each media source provides a type map representation of data types it is capable of producing. Likewise, the engine contains a set of default type maps that specify all the possible transformations that can be performed by the engine itself. A media source can throttle the amount of transformations it performs by reducing the size of its type map and letting the engine perform the rest using the default type map. By supplying comprehensive type maps encompassing multitudes of possible types within the engine, an underpowered media source that would normally be required to manipulate its data to arrive at the LUB type is relieved.

4 MediaBroker Implementation

MediaBroker (MB) architecture is implemented in two main pieces: an *engine* written in C, running on a cluster of machines or a single machine and the MB *client* library, also written in C, that clients can use anywhere in the distributed system.

Implemented on top of the D-Stampede distributed computing system [4], MB gains use of the D-Stampede abstractions that allow ordering of data in channels and provide streaming support with a uniform API on a number of computing platforms in the aforementioned hardware continuum. While D-Stampede itself is implemented on top of a reliable transport layer, our MB implementation contains very few bindings to D-Stampede, allowing portability across any transport framework as long as it offers stream-like semantics for transferring data and uniform API for multiple heterogeneous devices. Since D-Stampede inherits functionality from the Stampede system [5, 6], it includes a powerful clustering component that enables the MB engine to effectively use high-performance cluster machines for CPU-intensive transformations applied to high-bandwidth/low-latency streams appealing to co-located clients.

4.1 MediaBroker Engine

At the core of the engine is a *master* thread delegating connecting clients to *listener* threads. The listener thread then has the authority to instantiate data broker threads, notify data broker threads with information about new clients, and instantiate internal threads from dynamically loaded application code.

The internals of a data broker are shown in Figure 4. A Data Broker is a set of threads consisting of a *transport* thread, a *command* thread, a *LUB* thread, a *source_watch* thread, and a *sink_watch* thread. These threads are necessary because D-Stampede has no facility for polling multiple interfaces for the newest data [4]. The *databroker* thread transfers data from source to

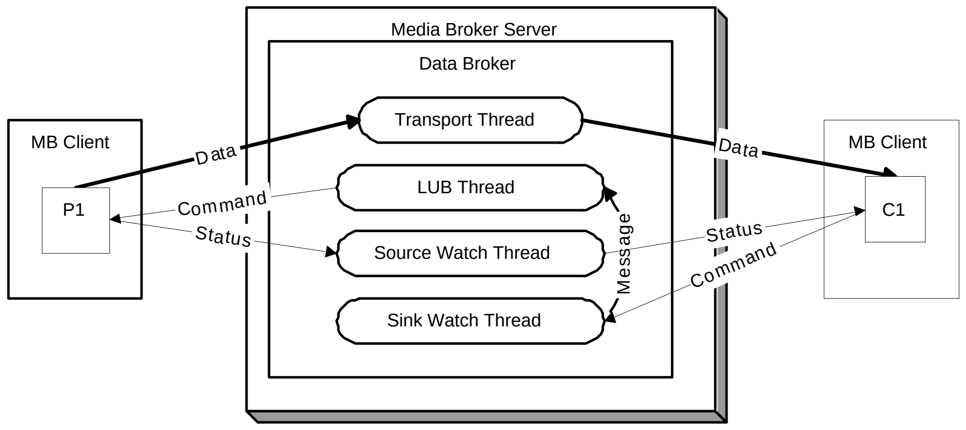

Fig. 4. Data Broker Internal Implementation

sink(s) while transforming data according to the transformation chain from the type map. The *command* thread blocks on the command channel and listens to the server's *listener* threads for messages about adding and removing sources and sinks. The *LUB* thread waits until a sink updates its status, calculates the optimal data type to request from the source and requests it. The *source_watch* and *sink_watch* threads block until the sinks or sources update their commands or statuses. The respective watch thread updates the appropriate structures of data transformations the *databroker* thread uses to transform data as it moves from the source to the sink.

4.2 Client Library

The MB client library is written on top of D-Stampede and leverages its client API. In fact, most calls map directly onto corresponding D-Stampede API calls. The library includes standard calls for initiating clients, media sources and sinks, and manipulating data types and type maps. Detailed discussion of the MB client library API [7] is beyond the scope of this article.

In order to satisfy the requirement for supporting heterogeneous devices, MediaBroker leverages D-Stampede's inherent cross platform support. As a result, we are able to support clients running on low-powered Compaq iPAQ handhelds with wireless connections alongside high-performance clusters for data manipulation.

4.3 Types and Type Maps

The MB type library is part of the client library as well as the MB engine. Types are implemented in C as simple structures, sets as linked lists, and types are associated with each other via memory pointers. The library also supplies functions for serialization of type maps for marshaling over the network.

The type system API that is part of the client library allows construction of types and type maps, comparison of individual types and type map traversal. The type system API also includes an algorithm for finding the LUB type.

Type modules are application-provided shared libraries, dynamically loadable with the dlopen() system call. At runtime, the type maps along with the corresponding transformation functions are loaded whenever media sources and media sinks might require their use.

A type module must implement the init_type_map() function, which is called when the module is loaded. It takes a pointer to an empty type map as an argument. This function must initialize the type map and add all types, their attributes, and their relationships to it. It may also optionally register type transformation functions across any type relationship (parent-child or siblings) and may do so in either direction.

Type transformation functions must also be implemented in the type module, using a standard interface. They take a pointer to a dynamically allocated buffer, and pointer to a buffer size value as arguments, and return an error code. To perform the transformation, the buffer may be modified destructively to contain the transformed data. However, since different data types often imply varying fidelity, they may also have data of different lengths. Therefore a pointer to the buffer is provided so that the buffer may also be reallocated to a different size, and a pointer to the size value is provided so that it may be destructively updated accordingly.

4.4 Name server/Directory Server

We use the OpenLDAP implementation of Lightweight Directory Access Protocol [8] as our name server. Features like readily available open source implementation, scalability, replication, enhanced security features, and the ability to extend existing schemas all lead to LDAP as a natural selection. Scalability of LDAP is particularly important given our application domain.

In the process of integrating the LDAP server into our runtime, it is our intention to remain name server-agnostic. A small C-to-LDAP interface library is provided for convenient translation from C structures to LDAP entries. Arguably very little effort would have to be expended to use an alternative name server implementation.

5 Performance Analysis

To understand the basic costs of our API calls, we present the results of μbenchmarks. Beyond API costs, we present end-to-end data movement costs relative to baseline TCP costs. To understand how our implementation is scalable with respect to the number of sources and sinks, we present similar end-to-end data movement costs for several scenarios. Finally, we analyze how fast our data broker implementation can respond to type adaptations in either a sink or a source. For each set of experiments, we show that our implementation provides reasonable support for our application space.

The name server runs on a RedHat Linux 2.4.18 kernel machine with a 2.0GHz Pentium IV processor, 512KB cache, and 512MB system memory. Except as noted, the tests run on RedHat Linux 2.4.18 SMP kernel machines with quad 2.2GHz Intel Xeon processors, 512KB cache, and 1GB shared memory. The test machines are connected with switched 100Mb ethernet. The name server is not located on the same switch, but it is on the same LAN.

Table 1
Normalized MediaBroker Benchmarks

API Call	Time (μseconds)	Std. Dev.
mb_init_producer	2575.16	459.32
mb_destroy_producer	2811.12	142.61
mb_init_consumer	2375.56	457.03
mb_destroy_consumer	2709.80	324.18

5.1 Micro-Measurements

Table 1 shows costs for *mb_init_producer*, *mb_destroy_producer*, *mb_init_consumer* and *mb_destroy_consumer*, that create and destroy media sources and sinks respectively. Although the name server API costs are low, we present "normalized" MB Client API costs independent of name server implementation. We do this "normalization" by subtracting the name server calls made during each MB API call. The costs depicted here are as expected from our implementation. For example, mb_init_source involves the creation of three threads and the allocation of four D-Stampede data abstractions. These Client API μbenchmarks demonstrate that the system supports applications that dynamically allocate and deallocate sources and sinks on a frequency of tens per second. This supports our application space, where EventWeb will allocate and deallocate source to sink streams at human speeds. The μbenchmarks for colocated clients are the same as for regular clients because both use the same out-of-band communication mechanisms.

5.2 End-to-End Data Movement Costs

To establish MB as a viable implementation to support streaming data from media sources to media sinks for our target applications, MB must exhibit the ability to move data with low latency and high throughput. In the experiments to follow, latency is half the time required for sending an item from the source to the sink and back, while throughput is measured in a system devoted entirely to streaming items from source to sink as quickly as possible. We vary the sizes of test items geometrically from 1 byte to 1 MB.

5.2.1 Isolating Engine Overhead

The majority of engine overhead is the data broker's routing and transforming of data from sources to sinks. To examine engine overhead, we first factor out the network and test the latency and throughput of communication on a single host. Three versions of data transfer within a single host are presented: (1) source to sink through MB where source and sink are both colocated clients running "internal" to the MB engine address space, (2) source to sink through MB where source and sink are running "externally," and (3) directly from source to sink transferring data solely over TCP/IP for baseline comparison.

Figure 5 shows the latencies and throughputs associated with various item sizes in the three test scenarios. As item sizes increase, MB latency increases faster than the baseline TCP latency because the MB API involves a round-trip time innate to the request-response semantic of the underlying Stampede API. The transmission of an item from an internal MB source to an internal MB sink involves two round trip times and data broker transport thread scheduling. These overheads are low relative to the tolerance of our application space to latency. For example, $\frac{1}{3}$ second of 22 KHz single channel 16-bit audio data is approximately 16 KB for which our measurements indicate a latency of 250 μseconds.

5.2.2 Engine Overhead with Network

By introducing the network into the previous experiment we hope to show that the overhead MB imposes on data transfer is minor relative to the overhead imposed by the limitations of network communications. We have run experiments based on two scenarios: (1) source to sink through MB across three machines, and (2) source to sink through a relay implemented solely on TCP/IP between three machines.

Figure 6 shows the latencies and throughputs associated with various item sizes in these two scenarios. When the network is introduced, our through-

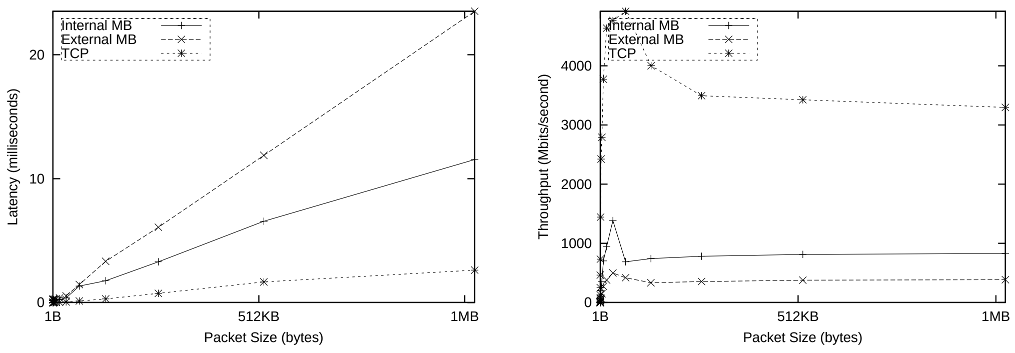

Fig. 5. Latency and Throughput of Data Transfer of Varying Sized Items Across a Single Host

put and latency measurements parallel the TCP baselines quite well. The aforementioned audio load is easily handled by our throughput capabilities. Furthermore, low resolution 30 fps uncompressed video formats fit within this bandwidth.

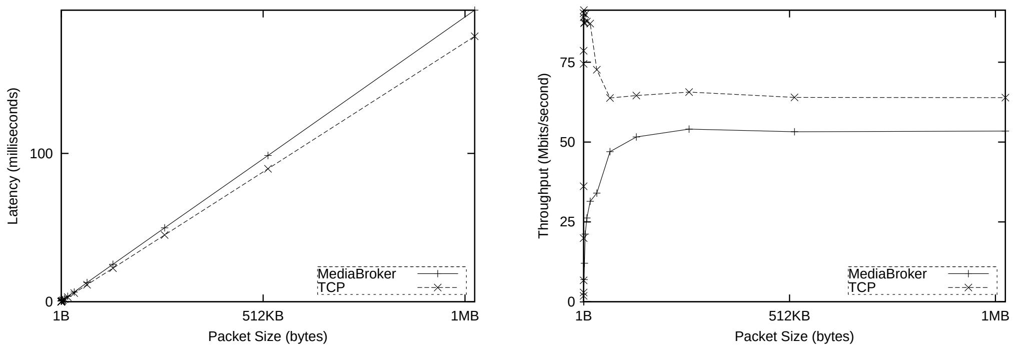

Fig. 6. Latency and Throughput of Data Transfer of Varying Sized Items Relayed Across Three Hosts

5.3 Testing Scalability

To test the scalability of our MB implementation, we present two experiments testing MB's ability to perform as numerous sources and sinks local to the engine host are connected into the system. The performance of transfer should degrade gracefully as limited processor and memory bus resources are shared between multiple sources and sinks.

5.3.1 Sink Stream Scaling

In situations where multiple sinks are drawing from a single source, we need to ensure that the system performance degrades gracefully as more sinks are

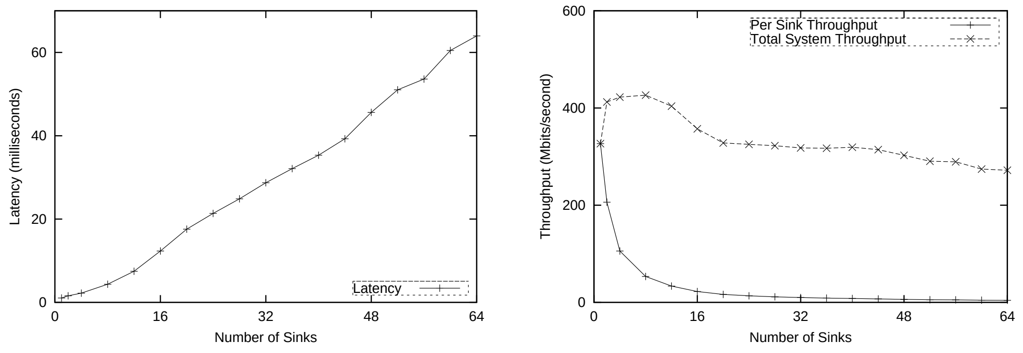

Fig. 7. Fan Out Scaling: Latency and Throughput of Data Transfer from a Single Source to a Varying Number of Sinks on a Single Host

added.

We run this experiment with a single source sending 16 KB packets to a varying number of sinks. The latency and throughput information is shown in Figure 7. In our application space, sources data needs to be shared by multiple sinks. For example, the several applications may want to share the media stream from a centrally located microphone. MB performance degrades gracefully as the data broker distributes to more sinks.

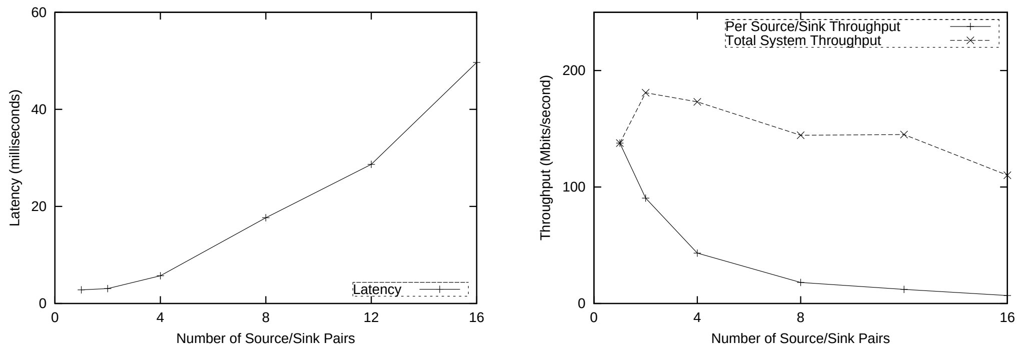

Fig. 8. Parallel Scaling: Latency and Throughput of Data Transfer from a Varying Number of Sources to an Equal Number of Sinks on a Single Uniprocessor Host

5.3.2 Data Broker Scaling

Our MB implementation must also scale as multiple data brokers are instantiated to serve media source data to media sinks. By instantiating n sources sending to n sinks on a single engine host, we hope to show linear performance degradation. We test a varying number of sources sending 16KB packets to distinct sinks. To present the results most clearly, the latency and throughput information shown in Figure 8 is the result of running this experiment on a machine with a single Pentium IV processor. The machine is equipped with

256 KB cache and 1 GB of memory. Again we find graceful linear degradation as source/sink pairs are added.

5.4 Type Adaptation Latency

We discussed in Section 3.2, the necessity to occasionally pause the data broker. We examine the latency imposed on data streams by this pause in the context of two different scenarios: (1) a media source changes its data type, and (2) a media sink requests a new data type. In order to perform these tests, we use NULL data transforming logic within the Data Broker to measure the latency of transfers from end to end.

To measure the adaptation latency of a media sink we instantiate a source and a sink. The sink requests a data type in the simple text type map shown in Figure 3(b), while the source produces data of the highest type. Every hundred iterations of the source/sink transfer, the sink randomly changes the data type it is requesting from the source in order to measure the time the data broker takes to recalculate transformation logic for the sink. The average latency resulting from this recalculation is 325.50 μseconds.

Similar to sink type adaptation, sources may change the data type that they produce. In order to isolate the time required to stabilize the system after a media source status change destabilizes it, we modify the sink type adaptation experiment so that the source randomly changes its produced data type and updates its status every hundred iterations. The average latency resulting from source type recalculation is 452.00 μseconds.

6 EventWeb: Demonstrating MediaBroker Utility

In this section we present a prototype implementation of EventWeb [9] using MediaBroker for distributed stream management and data transformation. EventWeb is intended to capture and correlate multimedia data in order to provide users with a meaningful, more refined view of the world around them. In response to queries from users, EventWeb employs feature extractors (such as video-based motion detection) to distill raw media into forms suitable as inputs to event detectors. Event detectors fuse multiple feature streams to detect domain events (such as an intrusion). MediaBroker's stream registration, discovery, transport, and transformation capabilities apply directly to EventWeb's feature extraction, enabling distributed distillation of raw media into features.

Consider a security system application implemented in EventWeb. It could simply query for live video from cameras when motion is detected. To enable this EventWeb application, several things are required. First, there must be a *Continuous Query Engine* (CQE) to accept, parse, and execute application queries to output resulting video segments. Second, one or more cameras must supply streaming video data. Third, a feature extractor is needed to infer motion data by analyzing video streams.

MediaBroker connects sources to feature extractors, and sources and feature extractors to the CQE. This is done dynamically in order to efficiently handle the changing requirements of the CQE, based on new queries that are made at different times. The implementation presented in Figure 9 provides these in a single camera environment.

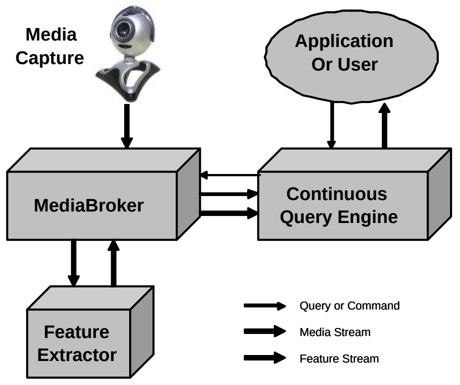

Fig. 9. EventWeb Using MediaBroker

The implementation of a camera source is simply a MediaBroker source client that produces video frames. The motion detection feature extractor is implemented as a type transformation — an edge in the video type map between the parent video frame type and the child motion detection type. Thus, the motion detection feature for a camera can be obtained by accessing the same camera source and requesting the motion detection type rather than the video frame type.

We leverage a prototype Continuous Query Engine (CQE), implemented to accept, parse, validate, and execute queries from an application against media streams and derived feature streams. We bind the feature extraction dispatch and result collection interfaces of the CQE to our MediaBroker implementation. To demonstrate simple single camera motion detection using our MediaBroker-enabled streaming and feature extraction, we begin by presenting the following query to the CQE after starting the camera source:

```
SELECT frame_num, content
FROM video1, mvFStream1
WHERE mv_pixel > 1500
```

In this query, the frame number (frame_num) and video frame image (content) are being returned from a media stream (video1) from MBCam. Also included in the query is mvFStream1, the feature stream from MBCam, which includes the field mv_pixel that represents how many pixels in the image have changed compared to the previous frame. The query returns results where the motion detection feature value is greater than a threshold, 1500 in this case.

Thus, the CQE needs both a stream of media frames from the camera, and a stream of motion feature data from the feature extractor in order to perform this query. The CQE is a sink to the MBCam source, but in addition to receiving the video frames from the camera, the CQE must also request the motion feature type from the type map. This will cause video frames from the camera to be processed by the type transformation, which is the motion detection feature extractor.

6.1 Results

The experimental setup involves three systems. The first includes the camera hardware and runs a MediaBroker source client. The second runs the MediaBroker, as well as the type module that contains the feature extractor type transformation. The third system runs the CQE, which includes the MB sinks, as well as the application code that displays the camera frames from the qualified query results. The camera is placed so initially only immobile objects are in the field of view, and then the MediaBroker, camera client, and CQE are started. The query is issued as printed above, and motion is generated in the camera's field of view at varying intervals, in the form of people moving. The sequence number, motion detection value, and query result (true/false whether a frame is returned) are recorded for each camera frame queried.

During the experiment there was one prolonged period of motion as well as several "spikes" of motion, which can be seen in the motion value data in Figure 10. Any time this motion value exceeded the threshold of 1500, the query returned the corresponding video frame (also shown in Figure 10).

These results show the efficacy of MB with one source and a simple query. The scalability and adaptability results presented earlier imply that MB will scale to larger EventWeb scenarios involving complex queries on multiple sources.

6.2 Performance and Scalability in EventWeb

One of the key aspects affecting performance in EventWeb is sinks that request different data types from the same source. Therefore it is important to under-

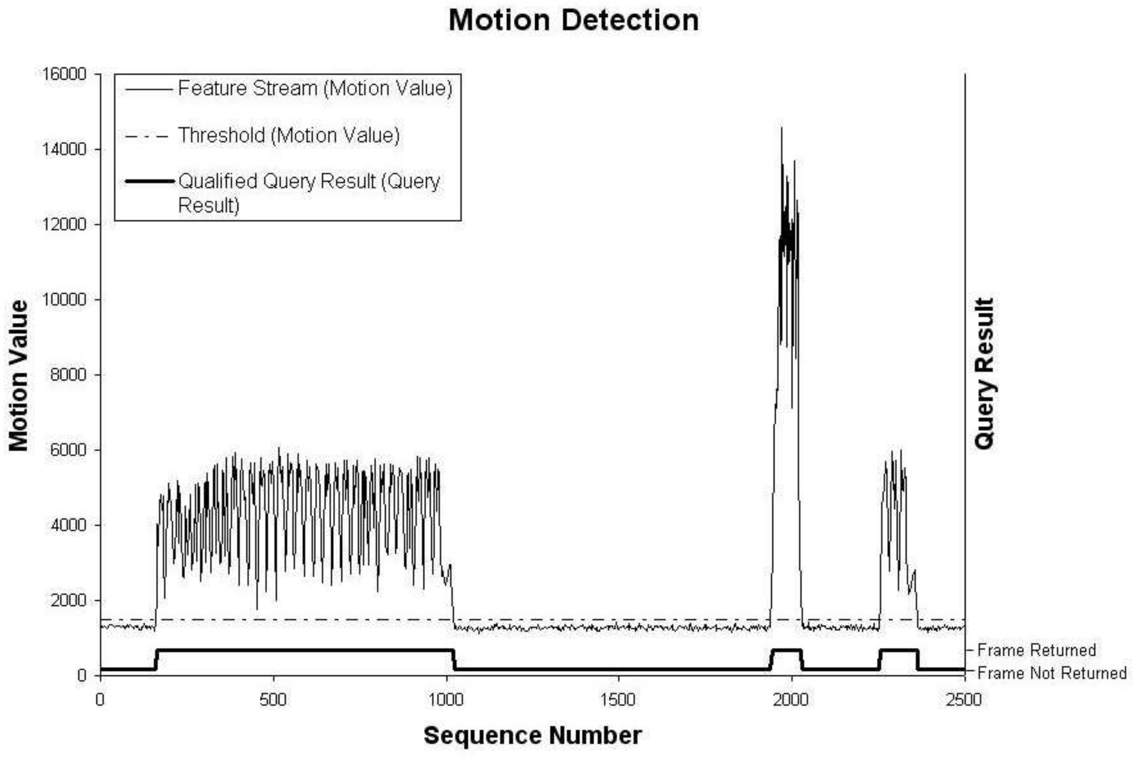

Fig. 10. Qualitative Experimental Results

stand the scalability of MediaBroker as more different data types are requested from a single source.

This experiment runs on a RedHat Linux 2.4.20 kernel machine with a single 1700MHz Intel Xeon processor, 256KB cache, and 512MB system memory. All processes are run on this system except the LDAP server which is run on the same system specified in section 5 for the name server.

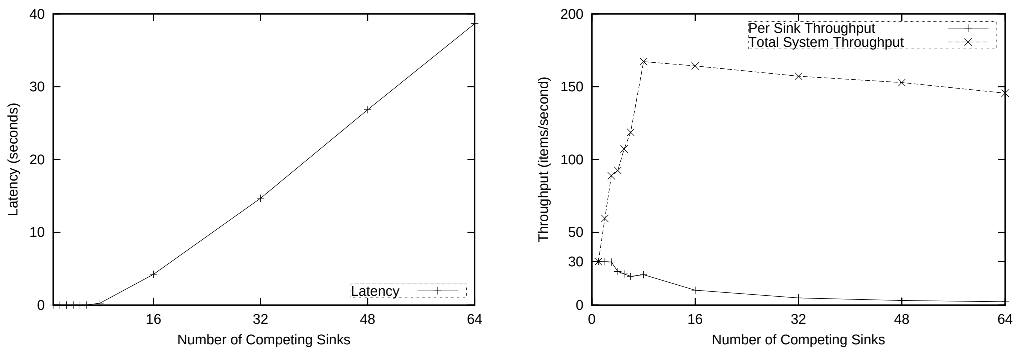

Fig. 11. Transformation Scaling: Latency and Throughput of Data Transfer to Competing Sinks

A source is created to produce video frames from a camera that are 176x144 pixels with 24-bit color depth. The type map used has the video frames as the root (highest) type, and 64 children of that type. Each of these children is an identical motion detection type with identical attributes, except for the name ("Motion1" through "Motion64"). The same motion detection feature extractor is registered as the type transformation function from the root node to each of its 64 children, thus a sink for any of the children data types will require a separate execution of the same transformation function. We vary the number of sink clients from 1 to 64, register each client to a different data type in the type map, and take end-to-end latency and throughput measurements as the number of competing clients varies.

Figure 11 shows when the system transitions from being producer-bound to being transformation-bound. At about 8 competing clients and fewer, the average latency is minimal and the total system throughput increases as more consumers are requesting streams. This appears because items are handled by the MediaBroker faster than the 30fps camera source is producing them. However as the MediaBroker must perform more transformations, it cannot handle each item as quickly so the transformations become the new limiting factor as we add more than 8 competing clients. Nevertheless, figure 11 shows that latency increases almost linearly and the throughput per sink degrades smoothly as more are added. The total throughput also drops gracefully as additional sinks incur more overhead. As expected, these are similar patterns to those exhibited in the Sink Stream Scaling experiment shown in figure 7.

7 Related Work

Stampede [5, 6] was first developed as a cluster parallel programming runtime for interactive multimedia applications. Stampede's focus is to provide efficient management of "temporally evolving" data, buffer management, inter-task synchronization, and the meeting of real-time constraints. Other features of Stampede include cluster-wide threads and consistent distributed shared objects. D-Stampede [4], extends Stampede programming semantics to a distributed environment spanning end devices (such as sensors) and back end clusters. However, Stampede is simply a temporally ordered transport mechanism, which neither recognizes the types of data flowing through the Stampede channels, nor has any built-in mechanism for type transformations.

Transcoding is usually performed either to suit specific requirements of a receiving device or to conserve network resources. The BARWAN project [10] shares some similarities with the MB project, targeting the exact set of requirements with the exception of a strong data-typing facility. Amir et al. [11] emphasize transcoding as a useful technique but do not have a facility similar to MB's extensible type system. Other works [12,13] focus on active networks, a concept similar to transcoding, albeit implemented at the IP level. In active networks the originator of a data packet can specify transcodings that need to be applied to the data en-route. MB applies this concept at a higher level of abstraction allowing execution of more complex transformations.

The Aura and Odyssey projects [1,14–16] aim at designing a comprehensive infrastructure for pervasive computing; they include provisions for runtime application adaptation to deal with unanticipated shortage of resources, type transformation, and a variety of other facilities for "masking uneven conditioning". Adaptation of application behavior to dynamic conditions is addressed in [17,18]. The focus of MB is different; it addresses the problem of overcoming the complexity of multi-format data transformations (similar to the TOM project [19]), while still allowing effective data sharing between multiple sinks. The latter implies that if multiple applications operate in a single smart space, MB facilitates the sharing of a device across applications commensurate with the advertised capabilities of the device.

The QoS Broker [20,21] uses a similar methodology to the MediaBroker. It takes QoS parameters from buyers and sellers (sinks and sources) and determines the least resources needed to satisfy all applications' QoS needs. This is analogous to the MediaBroker that takes type information from sinks and sources and determines the least (lowest) type needed to satisfy all applications' data type needs. Thus they use similar methods to meet different requirements in a pervasive computing setting: quality of service compared to data typing and transformation.

Type description languages have been recognized as a technique to structure and provide context to transcoding functions [22–24]. Nonetheless, we are not aware of any architecture that employs a type map approach to deal with transcoding complexity. Some work has been done in terms of constructing transcoding pipelines, specifically in the context of graphics subsystems [25, 26]. However, these mechanisms are currently confined to a single host.

Diverse application-level data sharing approaches have been proposed [11,27–30]. Proliferation of these can be explained, at least in part, by the difficulty associated with configuring and deploying de facto IP multicast on a typical IP network. Generally, we find this to be an open challenge, especially in the domain of pervasive computing.

8 Conclusions

MediaBroker (MB) is an architecture for the application domain of pervasive computing. We have motivated our architecture by examining the requirements of applications common to smart spaces. In particular, we have discussed the key features of this architecture: a type-aware data transport that is capable of transforming data, an extensible system for describing types of streaming data, and the interaction between the two. We have presented our prototype MB implementation's performance, and shown that it is sufficient for many pervasive computing streaming media applications. Finally, we have demonstrated the utility of MB for supporting scalable stream management and data transformation for feature extraction in a concrete EventWeb system example.

Although type maps are currently implemented as C data structures, future work may include using XML to represent type maps. This has several advantages. Marshalling type maps is made easy since XML data does not contain pointers and is trivially serializable. It also allows type information to be loaded from one or more XML files, so that dynamically loadable shared libraries are needed only for the type transformation functions. Finally, whereas currently a type must be specified precisely, using XML would facilitate adding search functionality, allowing types to be found based on their attributes.

8.1 Acknowledgements

The work has been funded in part by an NSF ITR grant CCR-01-21638, NSF grant CCR-99-72216, HP/Compaq Cambridge Research Lab, the Yamacraw project of the State of Georgia, 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 also thank Bikash Agarwalla for help with previous prototyped MB architecture design and implementation.

References

[1] M. Satyanarayanan, Pervasive computing: Vision and challenges, in: IEEE Personal Communications, 2001, pp. 10–17.

[2] R. Jain, Experiential computing, in: Communications of the ACM, Vol. 46, 2003, pp. 48–55.

[3] H. Ait-Kaci, R. S. Boyer, P. Lincoln, R. Nasr, Efficient implementation of lattice operations, Programming Languages and Systems 11 (1) (1989) 115–146.

[4] S. Adhikari, A. Paul, U. Ramachandran, D-Stampede: distributed programming system for ubiquitous computing, in: Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS), 2002, pp. 209–216.

[5] R. S. Nikhil, U. Ramachandran, J. M. Rehg, R. H. Halstead, J. C. F. Joerg, L. Kontothanassis, Stampede: A programming system for emerging scalable interactive multimedia applications, in: The 11th International Workshop on Languages and Compilers for Parallel Computing, 1998, p. 83.

[6] U. Ramachandran, R. S. Nikhil, N. Harel, J. M. Rehg, K. Knobe, Space-Time Memory: A parallel programming abstraction for interactive multimedia applications, in: Principles Practice of Parallel Programming, 1999, pp. 183–192.

[7] M. Modahl, I. Bagrak, M. Wolenetz, D. Lillethun, P. Hutto, U. Ramachandran, MediaBroker API: *http://www.cc.gatech.edu/ ˜davel/media_broker/api.h* (2004).

[8] W. Yeong, T. Howes, S. Kille, RFC 1777: Lightweight directory access protocol (March 1995).

[9] M. Modahl, I. Bagrak, M. Wolenetz, R. Jain, U. Ramachandran, An architecture for EventWeb, in: Proceedings of the 10th IEEE Workshop on the Future Trends of Distributed Computing Systems (FTDCS), 2004, pp. 95–101.

[10] E. A. Brewer, R. H. Katz, Y. Chawathe, S. D. Gribble, T. Hodes, G. Nguyen, M. Stemm, T. Henderson, E. Amir, H. Balakrishnan, A. Fox, V. N. Padmanabhan, S. Seshan, A network architecture for heterogeneous mobile computing, IEEE Personal Communications Magazine 5 (1998) 8–24.

[11] E. Amir, S. McCanne, H. Zhang, An application level video gateway, in: Proceedings of the third ACM international conference on Multimedia, 1995, pp. 255–265.

[12] D. L. Tennenhouse, D. J. Wetherall, Towards an active network architecture, ACM SIGCOMM Computer Communication Review 26 (2) (1996) 5–17.

[13] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, G. J. Minden, A survey of active network research, IEEE Communications Magazine 35 (1) (1997) 80–86.

[14] D. Garlan, D. P. Siewiorek, A. Smailagic, P. Steenkiste, Project Aura: Toward distraction-free pervasive computing, IEEE Pervasive Computing 1 (2002) 22–31.

[15] S.-W. Cheng, D. Garlan, B. R. Schmerl, J. P. Sousa, B. Spitznagel, P. Steenkiste, N. Hu, Software architecture-based adaptation for pervasive systems, in: ARCS, 2002, pp. 67–82.

[16] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, K. R. Walker, Agile application-aware adaptation for mobility, in: Sixteen ACM Symposium on Operating Systems Principles, 1997, pp. 276–287.

[17] E. K. Shankar R. Ponnekanti, Brad Johanson, A. Fox, Portability, extensibility and robustness in iROS, in: IEEE International Conference on Pervasive Computing and Communications (PerCom), 2003, pp. 11–19.

[18] E. de Lara, D. S. Wallach, W. Zwaenepoel, Puppeteer: Component-based adaptation for mobile computing, in: Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems, 2001, pp. 159–170.

[19] D. Garlan, TOM server main page: http://edison.srv.cs.cmu.edu:8001/ (2000).

[20] K. Nahrstedt, J. Smith, The QoS Broker, IEEE Multimedia 2 (1) (1995) 53–67.

[21] K. Nahrstedt, J. Smith, Design, implementation, and experiences with the OMEGA architecture, IEEE Journal on Selected Areas in Communications 17 (7) (1996) 1263–1279.

[22] K. Fisher, R. E. Gruber, PADS: Processing arbitrary data streams, in: Proceedings of Workshop on Management and Processing of Data Streams, 2003.

[23] P. J. McCann, S. Chandra, Packet types: Abstract specification of network protocol messages, in: Proceedings of ACM SIGCOMM, Vol. 30, 2000, pp. 321–333.

[24] G. Eisenhauer, The ECho event delivery system, ECho version 2.1 (July 2002).

[25] M. D. Pesce, Programming Microsoft DirectShow for Digital Video and Television, Microsoft Press, 2003.

[26] J. Knudsen, Java 2D Graphics, O'Reilly & Associates, 1999.

[27] Y. Chawathe, S. McCanne, E. Brewer, Scattercast: An architecture for internet broadcast distribution as an infrastructure service, Ph.D. thesis, UC Berkeley, 2000.

[28] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, J. W. O'Toole, Overcast: Reliable multicasting with an overlay network, in: Proceedings of the 4th Symposium on Operating System Design and Implementation, 2000, pp. 197–212.

[29] D. Pendarakis, S. Shi, D. Verma, M. Waldvogel, ALMI: An application level multicast infrastructure, in: Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS '01), 2001, pp. 49–60.

[30] Y.-H. Chu, S. G. Rao, H. Zhang, A case for end system multicast, in: ACM SIGMETRICS, 2000, pp. 1–12.