# MediaBroker:
# An Architecture for Pervasive Computing

Martin Modahl, Ilya Bagrak, Matthew Wolenetz,
Phillip Hutto, and Umakishore Ramachandran
College of Computing, Georgia Institute of Technology
801 Atlantic Drive, NW, Atlanta, GA 30332-0280, USA
[*mmodahl, reason, wolenetz, pwh, rama*]@cc.gatech.edu

## Abstract

*MediaBroker is a distributed framework designed to support pervasive computing applications. Specifically, the architecture consists of a transport engine and peripheral clients and addresses issues in scalability, data sharing, data transformation and platform heterogeneity. Key features of MediaBroker are a type-aware data transport that is capable of dynamically transforming data en route from source to sinks; an extensible system for describing types of streaming data; and the interaction between the transformation engine and the type system. Details of the MediaBroker architecture and implementation are presented in this paper. Through experimental study, we show 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.*

## 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 the 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 [22] domain.

Consider a *Family Intercom* application where sensors including microphones and speakers attached to desktop computers or wireless mobile devices are distributed throughout a smart space. User location tracking is performed by a series of cameras which relay location information to a user interface, which is also a form of sensor. Users actuate commands to connect speakers to microphone audio streams to initiate intercom-style conversations. With these disparate applications accessing similar resources, there should be a facility to share resources among them. A camera used to track people by the Family Intercom could be used to record the scene by another application.

Complex pervasive applications like Family Intercom require acquisition, processing, synthesis, and correlation of streaming high bandwidth data such as audio and video, as well as low bandwidth data from a user interface sensor [20]. Although each application may have distinct and non-overlapping 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 the corresponding APIs, to develop a variety of distributed applications.

In this paper, we present our contributions: (1) the MediaBroker architecture design addressing data typing, transformation and data sharing requirements, (2) an implementation of the proposed architecture and (3) performance analysis of our implementation.

The rest of the paper is organized as follows. We first discuss the requirements of target applications in Section 2. We present architecture in Section 3 and imple-

Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications (PERCOM'04)
0-7695-2090-1/04 $ 20.00 © 2004 IEEE

IEEE COMPUTER SOCIETY

mentation of the MediaBroker in Section 4. We present a survey of related work in Section 5. We analyze the performance of MediaBroker implementation in Section 6, and we conclude our findings in Section 7.

## 2. Requirements

Requirements for the MediaBroker architecture follow directly from the environment in which it operates and the sample application listed in the introduction. Applications in highly connected smart spaces encompass multitudes of sensors and demand scalability from the underlying infrastructure. High bandwidth audio and video streams that are part of these applications require system support for high throughput, low latency transport and a convenient stream abstraction. The 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. Therefore the core requirements are (1) scalability, (2) low latency and high throughput, and (3) adaptability.

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 transformation are (1) a common type language for describing transported data, (2) meaningful organization of transform functions applied to the data, (3) a flexible stream abstraction for data transfer that allows data sharing and (4) a uniform API spanning the entire hardware continuum.

### 2.1. Core Requirements

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 architecture 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. In terms of throughput, if data is produced faster than it can be transferred, then throughput becomes a bottleneck.

Adaptability entails several distinct requirements. First, it assumes the ability of devices to establish new and destroy existing communication channels at runtime while maintaining system consistency and stability. For instance, a user may choose to leave an intercom session 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 change of media sink requests.

### 2.2. A Common Type Language and Organization of Data Transforms

Consider an 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 the data streams that may be produced or consumed 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.

### 2.3. Underlying Transport Abstraction

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. In the case of video and audio streams, the data 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.

### 2.4. Providing Uniform, Flexible API

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 and architectures. In turn, a flexible API will enable a large number of applications to be built with minimum effort, while retaining the same data access semantics on a variety of platforms.


0-7695-2090-1/04 $ 20.00 © 2004 IEEE
IEEE COMPUTER SOCIETY

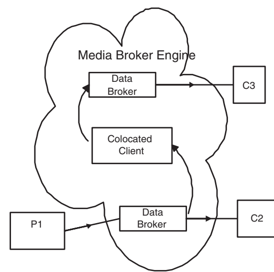


**Figure 1. Overview of MediaBroker Architecture**

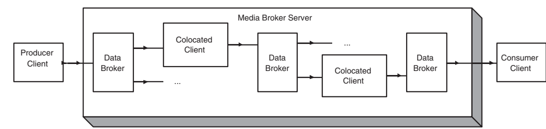


**Figure 2. Stacking Data Brokers and Clients**

# 3. The MediaBroker Architecture

## 3.1. Overview

The principal components of the MediaBroker (MB) architecture are the clients, the transport and transformation engine, its type system facility and a name server. The type system and name server form the foundation on which higher level components like the client and the engine 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 1 demonstrates basic data flow between both clients and transport engine. By making the client API available within the engine, we enable *colocated* clients to share computational resources of the engine. In the diagram, $P_1$ produces into the engine while $C_1$ 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 takes place when the MB engine requests from $P_1$ a type compatible with what both $C_1$ and 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 transports data from one media source to one or more media sinks. Essentially, a data broker satisfies the data sharing requirement outlined above. In addition, a data broker is 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. Figure 2 demonstrates stacking of colocated and distributed clients and data brokers.

**3.1.1. 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 1, 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 media 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. When a new sink is instantiated by a client, it is bound to an existing data broker.

**3.1.2. MediaBroker Engine** 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. The engine, upon client's connection, 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. At this point, non-volatile attributes describing the source of media such as its name are provided by the client to the engine. The listener then creates a data broker for the connecting media source, notifies the client of success, and supplies the source with a handle to its data broker. At the same time, the listener creates a mapping between resources that it allocates on behalf of a new producer, and stores the resources in the name server.

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 that they want to consume from. Using the source's name, the listener queries the name server to find the corresponding data broker. On success, it proceeds to notify the data broker of the addition of another media sink, communicating the data broker handle back to the newly attached sink.

**3.1.3. Data Broker** A high-level view of a data broker is presented in Figure 3. The data broker monitors the status of the attached media source as well as commands from all of its attached media sinks. We defer description of the type system until the next section, but both source status and sink commands are formatted


IEEE COMPUTER SOCIETY

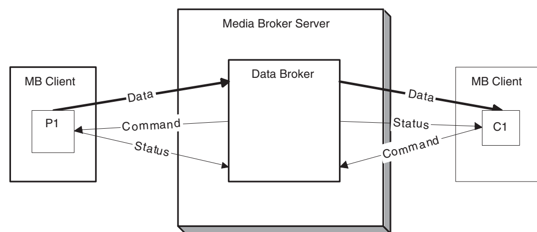


**Figure 3. Data Broker with Source and Sink**

as a data type. Status signifies the provided data type, while commands signify the data type requested. Thus, the data broker's goal is to rectify the source's status with the sinks' commands. The data broker achieves this goal 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. Essentially, the data brokers satisfy both the data sharing and data transformation requirements outlined in Section 2.

Channels used for type negotiation are depicted in Figure 3. We observe two distinct scenarios in which runtime adaptability of the system occurs. One scenario is prompted by a media sink stating, through a command, that it would like to consume data of some new type. The data broker then attempts adaptation by providing the new requested type to the media sink as soon as possible. Following a new media sink command, output to the media sink is paused while the new LUB type is calculated and appropriate transformations that need to be applied based on the new LUB type are determined. 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 6.4

**3.1.4. Type System** We define a convenient language for describing data types, their relations to each other and type transformations. A *type* is defined as a tuple of type domain and a set of type attributes. *Type domain* refers to a higher level description of type. For instance, all types of PCM audio data belong to the same domain. *Type attribute* consists of attribute name,

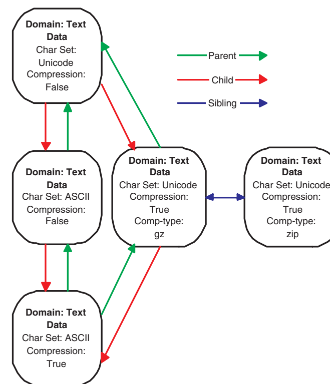


**Figure 4. Sample Type Map**

and an attribute value. *Attribute name* is an arbitrary ASCII string, and *attribute value* is the value of that attribute. Figure 4 shows several sample data types.

A *type map*, bearing strong resemblance to type lattices [2], is our data structure for describing relationships between various data types. Types are connected to each other via directed edges. A type map consists of an 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 4 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 4 supports this notion of quality as well. Edges connecting types signify that a data can be converted from one type to another.

The MB engine uses type maps extensively. 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 possi-


IEEE COMPUTER SOCIETY

ble 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.

**3.1.5. Name service** A name server is a crucial component of our framework, because it simplifies maintaining a mapping between producer and consumer names and computation resources and data structures associated with them at runtime. Since the functionality of many components depends on name server containing reliable information, write access to name server is restricted to the MB engine, while external clients have read access.

## 4. MediaBroker Implementation

### 4.1. Overview

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 [1], 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 [16, 21], 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 colocated clients.

**4.1.1. 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 5. A Data Broker is a set of threads consisting of a *transport* thread, a *command* thread, a *LUB* thread, a *source_watch* thread, and *sink_watch* thread. These threads are necessary because D-Stampede has no facility for polling multiple interfaces for the newest data [1]. The *databroker* thread transfers data from source to sink(s) while transforming as necessary. 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.

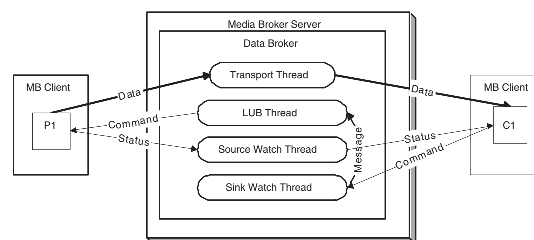


Figure 5. Data Broker Internal Implementation

**4.1.2. Client Library** The MB client library is written in C on top of D-Stampede. The library leverages the D-Stampede client API. In fact, most of the calls map directly onto corresponding D-Stampede API calls. The library includes standard calls for initiating clients, media sources and sinks as well as calls for manipulating type maps and type descriptions. Detailed discussion of the MB client library API is beyond the scope of this paper. [1]

**4.1.3. Types and Type Maps** The MB type library is written in C and is part of the client library as well as the MB engine. Types are implemented as simple C structures, sets are implemented as linked lists, and types are associated with each other via memory pointers. Since the data structure relies on pointers for traversal, the library also supplies functions for serialization/deserialization of type maps to a pointer-free architecture-neutral representation 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

1 MB API header file may be found at: *http://www.cc.gatech.edu/~mmodahl/media_broker/api.h*



IEEE COMPUTER SOCIETY

type system API also includes an algorithm for finding the LUB type.

Shared "type information" objects are dynamically loadable shared libraries. At runtime, the default type maps along with the corresponding transformation functions are loaded on demand whenever media sources and media sinks might require their use.

**4.1.4. Name server/Directory Server** We used the OpenLDAP implementation of Lightweight Directory Access Protocol [26] 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 was particularly important given our application domain.

In the process of integrating the LDAP server into our runtime, it was 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 port MB to an alternative name server implementation.

**4.1.5. Supporting Heterogeneity** When designing a system to provide a common interface to a continuum of devices, heterogeneity is paramount. Fortunately, D-Stampede, which serves as the core of this implementation instance, has been ported to several architectures, including ARM, which is popular in the embedded device space. 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.

## 5. Related Work

Stampede [16, 21] 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 [1], 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 [4] shares some similarities with the MB project, targeting the exact set of requirements with the exception of a strong data-typing facility. At the same time, [3] emphasizes transcoding as a useful technique but does not have facility similar to MB's extensible type system. Finally, [24, 25] 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 [6, 11, 17, 22] 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 has been addressed in [8, 23]. The focus of MB is different; it addresses the problem of overcoming the complexity of multi-format data transformations (similar to the TOM project [10]), 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.

Type description languages have been recognized as a technique to structure and provide context to transcoding functions [9, 15]. 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 [14, 19]. However, these mechanisms are currently confined to a single host.

Diverse application-level data sharing approaches have been proposed [3,5,7,13,18]. 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.

## 6. Performance Analysis

We present results of experiments testing our MB implementation in this section. First, to understand the basic costs of our API calls, we present the results of $\mu$-


IEEE COMPUTER SOCIETY

benchmarks. Beyond API costs, we present end-to-end data movement costs relative to baseline TCP costs. To understand how our implementation supports scalability with respect to 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.

For our tests, the LDAP name server runs on a machine with a 2.0 GHz Pentium IV processor with 512KB cache and 512MB memory. The name server runs RedHat Linux with a 2.4.18 Linux kernel. Except where noted, the tests run on machines equipped with quad 2.2 GHz Intel Xeon processors, each with 512 KB cache and access to the 1 GB of shared memory. The test machines run a RedHat Linux 2.4.18 SMP kernel installation. The test machines were connected with switched 100 Mb ethernet. The name server is not located on the same switch, but it is on the same LAN.

| API Call | Time ($\mu$-seconds) | 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 |

**Table 1. Normalized MediaBroker Benchmarks**

## 6.1. Micro-Measurements

In Table 1 we show costs for *mb_connect* and *mb_disconnect*, which connect and disconnect clients to the MB. We then examine *mb_init_source*, *mb_destroy_source*, *mb_init_sink* and *mb_destroy_sink*, 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 call time of 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 $\mu$-benchmarks 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 the Family Intercom will allocate and deallocate source to sink streams at human speeds. The $\mu$-benchmarks for colocated clients are the same as for regular clients because both use the same out-of-band communication mechanisms.

## 6.2. 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 exponentially from 1 byte to 1 MB.

**6.2.1. Isolating Engine Overhead** The majority of engine overhead is 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.

Figures 6 and 7 show the latencies and throughput 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 $\mu$-seconds.

**6.2.2. 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.


IEEE COMPUTER SOCIETY

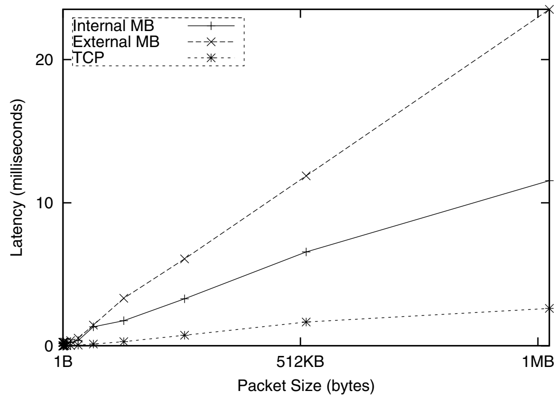


**Figure 6. Latency of Data Transfer of Varying Sized Items Across a Single Host**

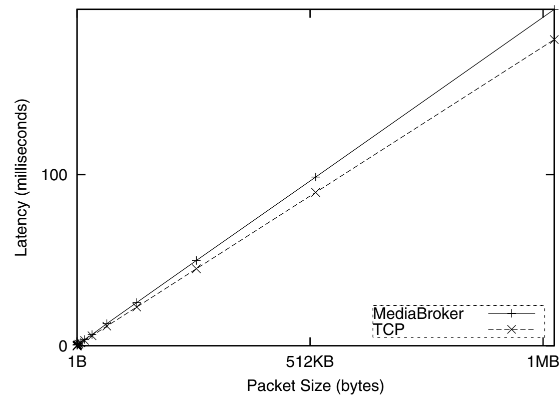


**Figure 8. Latency of Data Transfer of Varying Sized Items Relayed Across Three Hosts**

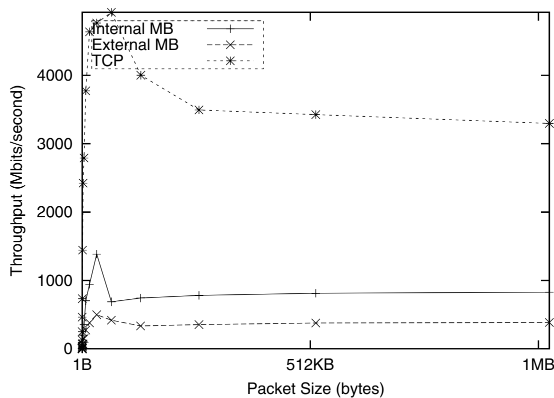


**Figure 7. Throughput of Data Transfer of Varying Sized Items Across a Single Host**

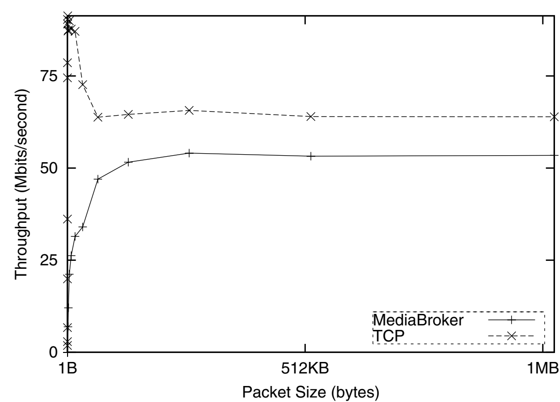


**Figure 9. Throughput of Data Transfer of Varying Sized Items Relayed Across Three Hosts**

Figures 8 and 9 show the latencies and throughput associated with various item sizes in the three test scenarios. When the network is introduced, our throughput 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.

## 6.3. 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.

**6.3.1. 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 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 10. In our application space, a 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.


IEEE COMPUTER SOCIETY

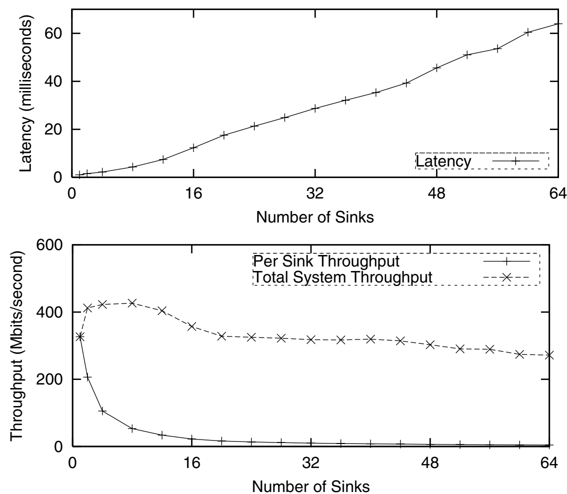


**Figure 10. Fan Out Scaling: Latency and Throughput of Data Transfer from a Single Source to n Sinks on a single host**

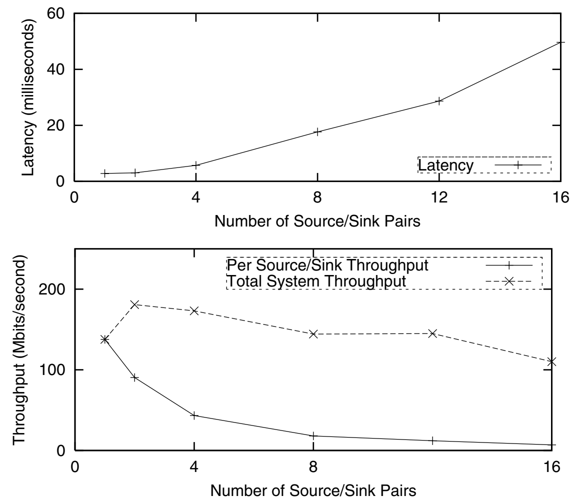


**Figure 11. Parallel Scaling: Latency and Throughput of Data Transfer from n Sources to n Sinks on a Single Uniprocessor Host**

**6.3.2. 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 11 are 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.

### 6.4. Type Adaptation Latency

We discussed in Section 3.1.3, 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 4, 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 $\mu$-seconds.

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 $\mu$-seconds.

## 7. 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. Finally, we have demonstrated that the performance of our implementation is sufficient for many streaming media applications.


IEEE COMPUTER SOCIETY

## 7.1. Acknowledgments

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] S. Adhikari, A. Paul, and U. Ramachandran. D-stampede: Distributed programming system for ubiquitous computing. In *Proceedings of the 22nd International Conference on Distributed Computing Systems(ICDCS)*, Vienna, July 2002.

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

[3] E. Amir, S. McCanne, and H. Zhang. An application level video gateway. In *Proceedings of ACM Multimedia*, San Francisco, CA, 1995.

[4] E. Brewer. A network architecture for heterogeneous mobile computing. *IEEE Personal Communications Magazine*, Oct. 1998.

[5] Y. Chawathe, S. McCanne, and E. Brewer. An architecture for internet content distribution as an infrastructure service, Feb. 2000. Unpublished work.

[6] S.-W. Cheng, D. Garlan, B. R. Schmerl, J. P. Sousa, B. Spitznagel, P. Steenkiste, and N. Hu. Software architecture-based adaptation for pervasive systems. In *ARCS*, pages 67–82, 2002.

[7] Y.-H. Chu, S. G. Rao, and H. Zhang. A case for end system multicast. In *Proceedings of ACM SIGMETRICS*, pages 1–12, Santa Clara, CA, June 2000. ACM.

[8] E. de Lara, D. S. Wallach, and W. Zwaenepoel. Puppeteer: Component-based adaptation for mobile computing. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, San Fransisco, CA, Mar. 2001.

[9] K. Fisher and R. E. Gruber. Pads: Processing arbitrary data streams. In *Proceedings of Workshop on Management and Processing of Data Streams*, San Diego, California, June 2003.

[10] D. Garlan. Tom server main page: http://edison.srv.cs.cmu.edu:8001/, 2000.

[11] D. Garlan, D. Siewiorek, A. Smailagic, and P. Steenkiste. Project aura: Toward distraction-free pervasive computing. *IEEE Pervasive computing*, 4:22–31, 2002.

[12] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, 1991.

[13] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. James W. O'Toole. Overcast: Reliable multicasting with an overlay network, 2000.

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

[15] P. J. McCann and S. Chandra. Packet types: Abstract specification of network protocol messages. In *Proceedings of ACM SIGCOMM*, Stockholm, Sweden, August 2000.

[16] R. S. Nikhil, U. Ramachandran, J. M. Rehg, R. H. Halstead, J. C. F. Joerg, and L. Kontothanassis. Stampede: A programming system for emerging scalable interactive multimedia applications. In *Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing*, 1998.

[17] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 276–287, Saint Malo, France, 1997.

[18] D. Pendarakis, S. Shi, D. Verma, and M. Waldvogel. ALMI: An application level multicast infrastructure. In *Proceedings of the 3rd USNIX Symposium on Internet Technologies and Systems (USITS '01)*, pages 49–60, San Francisco, CA, USA, Mar. 2001.

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

[20] U. Ramachandran, P. Hutto, B. Agarwalla, and M. Wolenetz. A system architecture for distributed control loop applications (extended abstract). In *Proceedings of the 9th International Workshop on Future Trends of Distributed Computing Systems(FTDCS)*, San Juan, Puerto Rico, May 2003.

[21] U. Ramachandran, R. S. Nikhil, N. Harel, J. M. Rehg, and K. Knobe. Space-time memory: A parallel programming abstraction for interactive multimedia applications. In *Proceedings of Principles and Practice of Parallel Programming*, pages 183–192, 1999.

[22] M. Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal Communications*, pages 10–17, Aug. 2001.

[23] E. K. Shankar R. Ponnekanti, Brad Johanson and A. Fox. Portability, extensibility and robustness in iros. In *Proceedings IEEE International Conference on Pervasive Computing and Communications (Percom 2003)*, Dallas-Fort Worth, TX, 2003.

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

[25] D. L. Tennenhouse and D. J. Wetherall. Towards an active network architecture. *Computer Communication Review*, 26(2), 1996.

[26] W. Yeong, T. Howes, and S. Kille. RFC 1777: Lightweight directory access protocol, Mar. 1995.



IEEE COMPUTER SOCIETY