Sytoplasm: A Middleware for Rich Sensor Networks

Arijit Ghosh

Tony Givargis

Technical Report CECS-09-01

April 14, 2009

Center for Embedded Computer Systems

University of California, Irvine

Irvine, CA 92697-3425, USA

(949) 824-8168

{arijitg, givargis}@cecs.uci.edu

Abstract

In this paper, we present the architecture of a middleware that supports rich sensor networks. A rich sensor network is composed of resource-constrained content-providing sensor devices connected to resource rich entities, for example desktops and laptops. The key design concerns of this system are flexibility, scalability, ease of use and provision for quality of service. We want the system to be autonomous; a device decides when to join the community and offer its content. Our middleware facilitates this process, allows users to lookup content and provides an efficient delivery mechanism of sensory content to the consumers. In this paper, we present a formal model of the system, the paradigm to program the system, the protocols for content request and techniques for content delivery. We propose network coding techniques combined with rate multiplexing for efficient usage of bandwidth.

Contents

List of Figures

Sytoplasm: A Middleware for Rich Sensor Networks

Arijit Ghosh, Tony Givargis

Center for Embedded Computer Systems

University of California, Irvine

Irvine, CA 92697-3425, USA

{arijitg,givargis}@cecs.uci.edu

http://www.cecs.uci.edu

Abstract

In this paper, we present the architecture of a middleware that supports rich sensor networks. A rich sensor network is composed of resource-constrained content-providing sensor devices connected to resource rich entities, for example desktops and laptops. The key design concerns of this system are flexibility, scalability, ease of use and provision for quality of service. We want the system to be autonomous; a device decides when to join the community and offer its content. Our middleware facilitates this process, allows users to lookup content and provides an efficient delivery mechanism of sensory content to the consumers. In this paper, we present a formal model of the system, the paradigm to program the system, the protocols for content request and techniques for content delivery. We propose network coding techniques combined with rate multiplexing for efficient usage of bandwidth.

1 Introduction

Sensor networks are taking real-time observation to unprecedented levels by embedding a network of sensor devices in the physical world. These devices, ranging from webcams to RFIDs, continuously monitor the environment. By extracting semantically rich information from these data, it is possible to *experience* any spatio-temporally displaced event. For example, webcams at a traffic intersection may continuously

record video. The data from these can then be combined to determine, if and when, an accident might have occured. Traditionally, sensornets have been viewed as resource-impoverished. The vision is that a large number of tiny networked nodes will cooperate for intelligent monitoring. However, recently there has been a push towards resource-rich sensornets. These typically have a heterogeneous architecture where the highly constrained nodes are supplemented by more powerful nodes, for example PDAs and laptops. These systems will be typically deployed in civilian spaces and can be used to support applications ranging from digital healthcare to real-world live searching. While similar in many ways to their poor cousins, the rich sensornets are more *general purpose* in flavor. This means a single infrastructure will be shared by multiple applications. A rich sensornet provides a shared infrastructure for delivering multimedia content captured by sensors to many users, possibly concurrently.

Rich sensornets open up the possibility of many exciting applications. One such application is *telepresence*. Telepresence refers to the ability of a person to experience the environment and events at a location other than his true location. For example, it allows one to watch a dunk in a basketball game as if he were standing under the hoop; oceanographers can study the world of deep sea without disturbing the marine ecology. Telepresence is a matter of degree. Telephone was one of the earliest technologies that enabled telepresence. Invention of television added a visual experience to the auditory experience. Bigger screen sizes, IMAX formats have further enhanced this. With the digitization of most multimedia technologies and omnipresence of the powerful Internet, there has been a surge of interest in this field with potential applications in diverse fields. Cisco Systems introduced TelePresence that creates unique, in-person experiences between people, places, and events in their work and personal lives over the network [1]. In the education sector, PEBBLES [2] provides a revolutionary solution for hospitalized, homebound and special needs children. PEBBLES connects children to their home classroom, allowing for total participation in classroom activities and complete social contact. Tele-immersion allows dancers in geographically distributed locations to collaborate [3]. One can easily imagine many uses in other industries, like advertising and entertainment.

Telepresence can be viewed as a geographically distributed nervous system. The nervous system of an animal receives sensory stimuli through the sensory nerves and transmits the data to the brain through the spinal chord. In telepresence, the sensory nerves correspond to the different sensing devices that capture

raw data. This is transmitted over the computer network, the spine, to the user, where the data can be processed to create user experiences. The architecture of this distributed nervous system has three layers: at the lowest layer is the data capture layer and consists of different kinds of sensing devices like cameras and microphones. The devices are connected directly to network enabled computers. The network of computers functions as the data delivery layer and resides in the middle of the architecture. At the highest level resides the telepresence application that combines data from many sensory sources to recreate the experience of the remote location. Figure 1 shows the architecture of such a system.

It is a well-known fact that there is a wide gulf between application programmers and the underlying system. This is even more true in sensornet applications where programmers are expected to be domain experts and not computer system experts. Application programmers thus need a simple but expressive view of the underlying system such that it is easy to compose applications from sensor streams. At the same time, the sytem should be able to efficiently manage its resources while serving the needs of the multiple applications. In order to facilitate both, we propose Sytoplasm, a middleware to manage rich sensornets. The application's view of the system is a network of addressable content. Content refers to unbounded streams of multimedia sensory data. It is not important how the content is generated. As such, physical level details are completely hidden from the user. Content is named and can be looked up and requested. At the heart of Sytoplasm is a content delivery system that will ensure that the right content is delivered to the right user. Interpreting the content and using it depends on the user and is **not** part of our infrastructure. In this paper, we present our design of Sytoplasm. The paper is organized as follows. In Section 2 we explore how our work is different from existing work. Section 3 presents our design in complete detail. In Section 4, we elaborate on the design of our content delivery subsystem. Finally, Section 5 concludes the paper by providing some future directions.

2 Related Work

Our design of Sytoplasm is inspired by the recent surge of interest in accessing live multimedia data over a network. There is a large body of work in this area. One one end of the spectrum are file-sharing applications and content delivery networks. On the other end are wireless sensor networks used for remote monitoring, sensing and even actuation. Here we review the general principles behind these approaches and refrain from

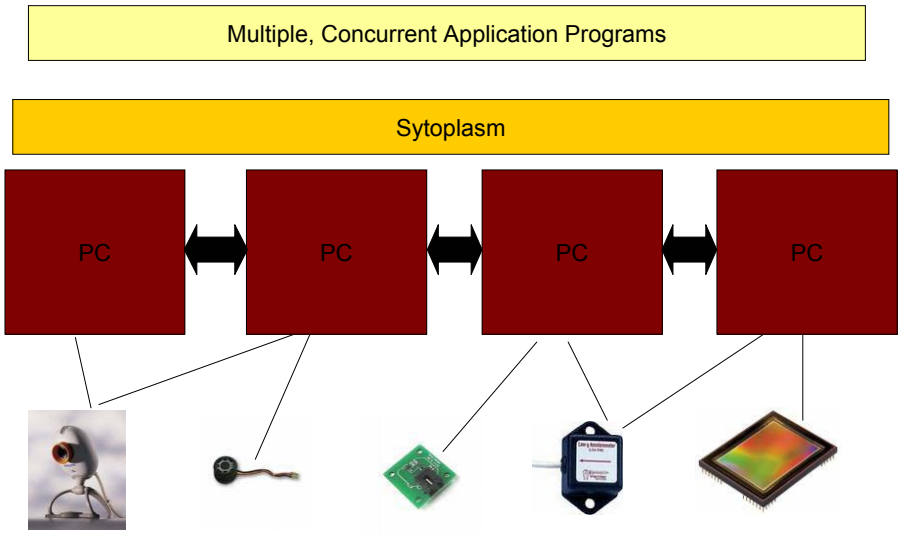

Figure 1: System Architecture

making an exhaustive survey in the interest of space.

File sharing networks: Sharing music, video and other forms of multimedia content over the Internet is very popular. Sharing data using BitTorrent [4] is so pervasive that it accounts for nearly 35% of all Internet traffic [5]. In general, BitTorrent distributes large files, called torrents, on separate networks. A torrent is broken down into a large number of smaller data blocks and these are shared independently. File sharing systems typically use a peer-to-peer overlay and distributes mostly static content.

Distributed event notification systems have also been built using peer-to-peer overlay networks. Corona [6] is a publish-subscribe system built on top of Pastry [7]. Users specify interest in specific URLs and updates are sent to users with instant message notifications. The polling overhead of monitoring URLs is distributed amongst cooperating peers and the overhead of crawling is amortized across several users interested in the same URL. This models a system where data is somewhat dynamic with content changing occasionally.

Real-time Stream Querying: The database community has focused on continuous query of real-time data streams originating from geographically dispersed locations. These include IrisNet [8], Borealis [9]

and Stream-Based Overlay Networks [10]. These systems provide a general purpose service for distributed query processing. They process dynamic data that has real time rate of change. However, they impose their own relational data model and define sophisticated operators. They do not address the underlying data delivery mechanism and can work as both pull and push based system.

Web Feeds: Web feeds, for example RSS [11], is a format used to publish frequently updated digital content, such as blogs, news feeds or podcasts. Users of RSS content use software programs called "feed readers" or "feed aggregators". The user subscribes to a feed by entering a link to the feed into the reader program. The reader can then check the user's subscribed feeds to see if any of those feeds have new content since the last time it checked, and if so, retrieve that content and present it to the user. Most pub-sub systems require publishers to change the way they generate and serve content, and require subscribers to register interest using private subscription formats. Recently, Cobra [12] has been proposed that interoperates seamlessly with existing RSS sources. These systems work with data that have an unpredictable rate of change. Most systems use a client-server architecture that follows a publish-subscribe model.

Distributed Publish-Subscribe Systems: Pub-sub systems can be divided into topic-based and content-based. In the former [13][14][15], subscribers register interests in a set of specific topics. Producers that generate content related to the topics, publish the content on the related *topic channel*. Subscribers receive asynchronous updates via these channels. Such systems requires both publishers and subscribers to decide upfront on what topics will be published in each channel. The alternative is *content-based* systems [16][17] where subscribers express their interest in terms of content attributes. Pub-sub systems have a pull-based architecture where data changes slowly over a period of time.

Content Distribution Network: Content distribution networks are dedicated collections of servers located strategically across the wide-area Internet. Content distribution networks operate by forming an overlay topology consisting of the content provider and content subscriber. A number of application-level multicast systems [18] have been built using DHTs that construct an information dissemniation tree with the multicast group members as leaf nodes. The multicast tree is formed by joining the routes from the leaves to the root. Akamai [19] is the most popular provider of content distribution infrastructure that is used by popular news and sports websites. The content is proactively pushed to the numerous CDN servers. The architecture relies on DNS interposition or URL rewriting at origin servers to direct requests to the nearest

Table 1: Comparison of content delivery systems

	Dynamism	Liveness	Data Model	Architecture	Push/Pull	Multi-user
File sharing	Static	-	-	P2P	Pull	Yes
Stream Query	Dynamic	Variable	Yes	-	-	Yes
Web feeds	Dynamic	Slow	Maybe	Both	Pull	Yes
Pub-sub	Dynamic	Slow	Yes	P2P	Pull	Yes
CDN	Dynamic	Flash crowd	No	C/S	Push	Yes
Sensor Network	Dynamic	High	No	P2P	Pull	No
Sytoplasm	Dynamic	High	No	P2P	Pull	Yes

CDN replica.

Sensor Networks: Sensor networks are a special kind of distrubtion network for real time sensed data. Typical sensor networks are different from other kind of systems discussed above. Sensor networks are often severely resource limited, connected by unpredictable wireless links and are usually deployed in harsh and difficult-to-reach environments. Sensor networks continuously monitor the immediate environment and hence generate high volumes of continuous data. Sensor networks also employ a pub-sub mechanism [20] where relevant data is filtered and aggregated as content flows back to the user. However, owing to the stringent resource availability, sensor networks are typically custom built and is not general purpose. They cannot support multiple concurrent applications that differ vastly in their content needs.

Sytoplasm is similar in spirit to all of the above approaches. The objective is to distribute multimedia content from servers to clients. However, our premise and requirements are a unique combination of some of the above-mentioned features. Our content servers are different kinds of sensors, that continuously monitor the surrounding and generate *live* data. The content is inherently heterogeneous as the sensors could be very different from each other. We want to promote autonomy, in the sense that content comes and goes in a ad hoc manner. Any number of clients can request data from any number of servers. Owing to volume and real-time nature of the data, it is not possible to push content to servers in anticipation of future demand. Since we are concerned only with content delivery, we do not want to impose any special data modeling requirements. Finally, in the interest of scalability, we cannot rely on a strict client server model. Table 1 positions our work relative to existing work.

3 Sytoplasm

The Sytoplasm middleware is intended to run on a network of sensory devices generating live, continuous data. Its objective is ease of development of applications that are composed of dynamic multimedia data from geographically distributed sources, for example telepresence applications. As such, Sytoplasm was designed with the following objectives in mind.

- **Data-centric view:** Programmers need an intuitive way of modeling and reasoning the system. A sensornet produces multiple time series data sequences produced by webcams and sensors. This data is the primary entity of importance to an application programmer. A programmer composes an application data by combining and performing feature extraction on this unbounded data. A programmer should operate at a higher level of abstraction and not be concerned about content generation and delivery. Hence a programming paradigm is necessary that will allow easy specification of an application in terms of data and its associated transformations.

- **Autonomy:** Inspired the success behind the Web model, we want our rich sensornets to be flexible enough to make content generators decide when to join the network and if to make their content available. Thus both producers and consumers are granted autonomy in their actions. A powerful effect of this is that sophisticated content can be generated by combining existing content and the former itself can be made available to other consumers for use.

- **Scalability:** Availability of content and demand for the same is very un[redictable. As such, an equally important requirement is for the system to scale easily to handle many concurrent providers and consumers.

3.1 Overview

The sensors in a sensornet continuously sense the surrounding environment. Each sensing device, hence, produces an unbounded stream of data and is connected to other devices. Based on this, Sytoplasm treats the world as a network of sensors, each generating one or many data streams. A stream corresponds to sensory data and is a sequence of timed sensor samples. Each stream is distinguished from every other stream in terms of a few basic properties, for example, location. A program running on a client node logically views

the system as a collection of streams. The programmer can ask for any number of streams. Sytoplasm ensures efficient delivery of all streams. A programmer can then combine the streams in an application-specific way to extract the desired functionality out of the sensor network. Sytoplasm thus serves two valuable purposes: one, it provides a uniform data-centric view of the system in terms of sensor streams; second it provides an efficient stream-delivery substrate.

The stream-oriented paradigm greatly simplifies sensornet application development. However, the real power of Sytoplasm comes from the ***extensibility*** of the system and ***versatility*** of stream semantics. Sytoplasm allows any sensor to get *online* and provide its stream to the world as long as it follows a protocol. Similarly, a client node, by following a certain protocol, can access any streams that are online. Thus, extensibility is embodied at both ends of the service infrastructure: adding and deleting content(streams) at will on end; and allowing any number of clients to access any content on the other end. As will be explained later, Sytoplasm makes no attempt in interpreting the data in streams. This allows streams to be extremely versatile and represent any kind of content. For example, imagine a camera sensor generating a stream consisting of N frames/second. Typically, accessing and delivering this stream requires heavy use of resources. Provided the client's requirment is not violated, a new stream can instead be derived which compresses $M(\leq N)$ frames into one data element. Sytoplasm will still view this new stream as all other streams and allow it to get online and be accessed as long as the protocol is adhered to. In this way, Sytoplasm allows a data element of a stream to be a stream in itself. Alternatively, an application program can itself generate a stream of data by combining streams from the network. The programmer, if she so chooses, can make this stream available to the world. The data type of this stream is completely defined by the programmer. An example is a stream of all accidents in the city of Los Angeles which is derived from the streams generated by cameras at all traffic intersections. The first responders can use the *accident* stream for their own application.

Streams with a one-one correspondence with sensing devices are called *inherent* streams. A stream built from other streams with richer semantics is called a *composite* stream. Sytoplasm thus makes the sensornet a powerful content provider by allowing intuitive creation of composite streams and easy addition of both inherent and composite streams to the underlying system.

3.2 System Architecture

The architecture of the system consists of three kinds of logical nodes attached to a networking infrastructure. The networking infrastructure could be wired or wireless, running a connection-oriented protocol. The three types of nodes include the following.

- **Producers**: These are the nodes that generate streams. A producer can generate either an inherent stream, typically when a sensing device is attached to it, or a composite stream. A producer can generate any number as well as both kinds of streams. A producer can have any hardware running any operating system. The producer has to adhere to certain specification to connect to the network. However, that is beyond the scope of this document.

- **Consumers**: These are the nodes that request Sytoplasm for a composite/inherent stream and consume it when delivered. Typically, these will be the nodes that will be running the application. Similar to a producer, a consumer can run on a unique local system and connects to the underlying network by using a standard protocol.

- **Server**: This is the node that runs Sytoplasm. It handles registration of new streams, consumer requests and delivery of streams. The server shall make no attempt to interpret the data delivered. The server should also validate stream requests made by consumers. The server should be able to handle multiple, concurrent requests to the same stream. The server shall honor data integrity and stream security. It provides efficient resource usage and tries to accomodate application quality requirements.

The above architecture is purely logical in nature and provides no indication of the mapping between the logical and physical nodes. For example, the same physical node may act as both the consumer and producer. Similarly, Sytoplasm might be distributed over multiple physical nodes, where some/all of them could be a consumer and/or producer.

3.3 System Operation

3.3.1 Streams and Feeds

A stream is an uninterpreted sequence of raw data and has the following characteristics.

- A stream is an unbounded sequence of data. Each data element is called a sample.
- The sampling rate of a sensor is fixed. This implies that as long as the sensor is operational, samples will be generated at fixed intervals.
- Unless stored explicitly, data in a stream is lost forever and cannot be reproduced. The responsibility of storage lies with the user.
- The data elements in the stream arrives *online* and are not available for random access from disk or main memory.
- A stream has some basic properties like location and sampling rate that are important to the system for content delivery. A stream can have some additional properties which does not have to be exported to the system.

A stream is formally defined as follows.

Definition 1. A **stream** $\mathbb{S}$ is an unbounded sequence of data, $\mathbb{S}=\langle s_1, s_2, \dots \rangle$. Each s_k is an uniterpreted data element.

Let $\mathbb{R}^+$ denote the set of positive real numbers. Let $\mathcal{T}$ be a discrete, ordered time domain. Then associated with each feed are the folowing mappings.

Definition 2. A **width** Γ is a mapping of a stream to the width of a single sample in the stream.

$$\Gamma \quad : \quad \mathbb{S} \rightarrow \mathbb{R}^+$$

Definition 3. A **rate** Λ is a mapping which shows the rate at which a stream generates data.

$$\Lambda \quad : \quad \mathbb{S} \rightarrow \mathbb{R}^+$$

Definition 4. A **start** Ψ is a mapping which indicates the time at which the stream started generating data.

$$\Psi \quad : \quad \mathbb{S} \rightarrow \mathcal{T}$$

As long as a stream is online, data will be continuously generated. Most of the times, a consumer will be interested in data that spans a certain finite period. To represent this, we introduce the concept of a *feed*. A feed is a bounded instance of a stream. In other words, it is a snapshot of a stream for a certain window of time. A feed is always associsted with one and only one stream. A stream can have multiple feeds associated with it. Different feeds of the same stream can belong to the same or different applications.

A feed $\mathbb{F}$ refers to a stream with samples within a certain time period. Let $\mathcal{S}$ be the set of all streams in the system. Then formally,

Definition 5. A **feed** $\mathbb{F}$ is a bounded sequence of uninterpreted data, $\mathbb{F}=\langle s_1, s_2, \ldots s_k \rangle$. Each sample, $s_i \in \mathbb{S}_j$, where $\mathbb{S}_j \in \mathcal{S}$.

Each feed is associated with a time window, Δ with a start time of Δ_s and end time of Δ_e.

Definition 6. A **window start** Δ_s is a mapping which indicates the start of the time interval for which the feed is of interest.

$$\Delta_s \quad : \quad \mathbb{F} \rightarrow \mathbb{R}^+$$

A **window end** Δ_e is a mapping which indicates the end of the time interval for which the feed is of interest.

$$\Delta_e \quad : \quad \mathbb{F} \rightarrow \mathbb{R}^+$$

In addition, if $\mathbb{F}$ corresponds to stream $\mathbb{S}$, then

$$\Psi(S) \leq \Delta_s \leq \Delta_e$$

Finally, a consumer may request that a feed be delivered with a certain average *jitter*. Jitter, δ, is the variation of the time stamp of a sample. The average jitter, δ_{mean} is the mean of the jitters of all samples in a feed. Then we have,

Definition 7. The **average jitter**, δ_{mean}, is a mapping which computes the mean of the jitters of individual

samples in a feed, $\mathbb{F}$.

$$\delta_{mean} \quad : \quad \mathbb{F} \rightarrow \mathbb{R}^{+}$$

3.4 Protocol

The design and functionality of Sytoplasm is based on an underlying protocol. The *stream transfer protocol* (STP) is an application-level protocol with the flexibility and extensibility to accomodate and scale to thousands of sensor streams. It is a generic, stateful, object-oriented protocol which can be used for easy addition and accessibility of sensor content on a networked system. A key feature of STP is its agonsticism to the type of data being transferred which enables systems to be built independently.

The protocol is centered around the three basic operations that entities connected to Sytoplasm perform: producers *register* their streams, consumers *demand* for feeds and Sytoplasm *deliver*s feeds. STP is a message passing protocol where messages are classified either as requests or responses. A message has a well-defined structure: a header followed optionally by a body. We now explain the basic elements of the protocol based around the three supported operations.

Registration: In order to make itself available to other applications, a stream has to register itself with Sytoplasm by exporting its *name* which is mapped to a globally unique identifier by Sytoplasm, the *location* in space according to a well-defined convention, the *size* of each sample in the stream, the *rate* at which the sample are generated and the *start time* at which the stream comes online.

Feed lookup: A consumer looks up for streams that match a specific criteria by sending a request to Sytoplasm. The consumer specifies a predicate which could be the name of the stream or a subset of its characteristics, for example location and rate. Sytoplasm responds with a list of matching streams.

Feed request: To request for a feed, a consumer has to specify the name, location, start and end times and the mean jitter. The server verifies that the stream exists in the system. It then checks to see if an existing feed's window subsumes the current request. If yes, then it does not send any new data request to the producer. If a feed exists but the windows do not match, then the server sends a window adjustment message to the producer. If no feed exists from the requested stream, the server sends a request to the producer for data by specifying the time window. A producer responds by sending samples to the server.

The server has to make adjustments between the requested and available data rates. If this is the first sample sent by the server to the consumer, the server sets up a feed session and generates a unique session identifier. For every sample, the server also computes the jitter. Finally, either the consumer or the server can terminate a feed seesion by sending the following.

3.5 Programming Paradigm

Programming the system is based on three simple concepts: *streams*, *feeds* and *kernel*. We have already explained the concept of streams and feeds. A kernel is a computational entity that takes in zero or more streams/feeds and generates zero or more composite streams.

A programmer programs the system by building a **stream flow diagram** which is similar to a data flow diagram. A SFD interconnects kernels and streams/feeds with arcs that act as communication links. The SFD allows the programmer to specify the type of data that is of interest, the time instants when the data is required and the processing associated with it. The output of a SFD is a composite stream. This is a key concept as it allows ***composability***. Thus a subsystem (or a subgraph) can be abstracted as a composite stream. This allows reuse and sharing which are key to scalability and ease of use. For example, a subsystem can correspond to a multiplexer with N data feeds and one control feed. When named and made available through the registry, it could be used by other application programs. We call this model the content oriented programming paradigm for sensor networks. Figure **??** shows a programmer's view of the real world web.

We now formally define the **Stream Flow Graph** (SFG). A SFG is similar to a data flow graph. The primary differences are in the semantics of the firing operation and the concept of data store.

Definition 8. A stream flow graph is a bipartite labeled graph where the two types of nodes are called *kernels* and *streams*. We use the latter in a generic sense from here on and can refer to an inherent stream, composite steram or a feed.

$$
\begin{aligned}
G &= \langle \mathcal{K} \cup \mathcal{S}, E \rangle \quad where \\
\mathcal{S} &= \{\mathbb{S}_1, \mathbb{S}_2, \ldots \mathbb{S}_n\} \quad is\,the\,set\,of\,streams \\
\mathcal{K} &= \{k_1, k_2, \ldots, k_m\} \quad is\,the\,set\,of\,kernels \\
E &\subseteq (\mathcal{S} \times \mathcal{K}) \cup (\mathcal{K} \times \mathcal{S}) \quad is\,the\,set\,of\,edges.
\end{aligned}
$$

Kernels represent the operations. Streams (and feeds) are an abstraction of data and are not stored unless explicitly requested by an application with a finite but bounded latency. Each SFG has one or more output streams which can act as inputs to other SFGs. This attribute highlights the composability of the system. Streams can be further subdivided into *control* and *data* streams. Control streams are introduced to indicate the presence of sequence control; certain kernels are enabled only when the right control values appear on the control stream. Edges are channels of synchronous communication with finite delay and no storage capabilities. S is a subset of streams called the *starting set*; these streams represent inputs to a stream flow graph (or subgraph).

$$S = \{\mathbb{S} \in \mathcal{S}, |(k,\mathbb{S}) \notin E, \forall k \in \mathcal{K}\}$$

T is a subset of streams called the *terminating set*; these represent outputs from a stream flow graph (or subgraph).

$$T = \{\mathbb{S} \in \mathcal{S}, |(\mathbb{S},k) \notin E, \forall k \in \mathcal{K}\}$$

The set of input streams to a kernel k, and output streams from a kernel k are denoted by $I(k)$ and $O(k)$.

$$I = \{\mathbb{S} \in \mathcal{S}, |(\mathbb{S},k) \in E\}$$
$$O = \{\mathbb{S} \in \mathcal{S}, |(k,\mathbb{S}) \in E\}$$

Throughout this paper, we refer to our model as the uninterpreted stream flow graph. This is because the actual semantics of the data in the streams are not relevant for the system functionality. The presence of tokens(or samples) of known size in the streams is all that is necessary to trigger the nodes. In case the streams are purely data feeds, this corresponds to a the classical synchronous data flow model. The presence of control streams allow programmers to determine when the corresponding kernel will be fired.

The above description creates an impression that raw data from the sensory sources are delivered to the user. This is partially true since our system provides a content registration and delivery architecture and

does not provide a computation platform. This, at first glance, seems contrary to conventional sensornet wisdom where computation is pushed to the data source to reduce resource requirements. We note, that pushing computation in the network is the same as mapping kernels to nodes in our system. As long as our system knows where the location of the producer (stream) and the consumer(the kernel), it can handle efficient delivery. The kernel mapping problem is assumed to be handled at a higher layer. While our system does not handle kernel mapping, it also does not inhibit kernel mapping.

3.5.1 Firing and Firing Semantic Sets

Classical synchronous data flow graphs follow the simple model that an operation is fired if all the input edges contain tokens and no tokens are present on output edges. We use the term token to refer to a sample. As pointed out in [], this control mechanism works only when operations are primitive and fail when they are complex or other subgraphs. Accordingly, we follow a firing semantic introduced by the same authors. The (input) firing semantic set refers to a subset of input edges that must contain tokens to enable the node. Similarly, an output semantic set can be defined as the subset of output arcs that must be empty.

Definition 9. A marking is a mapping

$$M : \mathcal{S} \rightarrow \{0,1\}.$$

A token is available at stream $\mathbb{S}$ at current time instant, τ_{now}. In other words, $\mathbb{S}(\leftarrow,\tau_{now}, \tau_{now})$ returns a non-empty tuple. A stream is said to have a token available in marking M if $M(\mathbb{S})$=1. An *initial marking* M_0 is a marking in which a subset of the starting set of streams contain tokens. A *terminal marking* M_t is a marking in which a subset of the terminating set of streams contain tokens.

Associated with each kernel are two sets of streams called *input firing semantic set* S_1 and *output firing semantic set* S_2.

$$\begin{aligned} S_1(k,M) &\subseteq I(k) \\ S_2(k,M) &\subseteq O(k) \end{aligned}$$

The input firing semantic set refers to the subset of input streams that must have a token available for the kernel to fire. The output firing semantic set refers to the subset of output streams that receive a token when the corresponding kernel is fired.

Definition 10. A *firing* is a partial mapping from markings to markings. A kernel k is firable at marking M if the following conditions hold

$$M(\mathbb{S}) = 1 \quad for\,all \quad \mathbb{S} \in S_1(k,M)$$

When the kernel is fired, tokens from the firing set $S_1(k, M)$ of streams are deleted using the operation (−,t). New tokens are placed on each stream belonging to the output firing semantic set $S_2(k, M)$ using the operation (+,t). Note the above mapping does not specify that the output firing semantic set has to be empty. This is different from a conventional data flow graph. Thus, a new marking *M'*, resulting from the firing of a kernel *k* can be derived as follows:

$$M'(\mathbb{S}) = \begin{cases} 0 & if\ \mathbb{S} \in S_1(k,M)\ and\ \mathbb{S} \notin S_2(k,M) \\ 1 & if\ \mathbb{S} \in S_2(k,M) \\ M(\mathbb{S}) & otherwise \end{cases}$$

A firing sequence is a sequence of kernels in the order in which they were fired. When fired concurrently, the order is non-deterministic.

4 Runtime System

The runtime system is responsible for enabling the *plug-and-use* paradigm for sensornets, support multiple concuurent applications and manage resources efficiently. The system accepts requests for feeds, locates the corresponding stream and delivers the feed to the user while providing some quality of requirements guarantees. Providing a desired quality of service to all concurrent applications requires clever management of underlying limited resources. In this section, we first look at the overall design. We then explain in detail the design of one of the modules. Finally, we present some preliminary evaluation results.

4.1 Design

The runtime system is designed to provide correct functionality, as specified by a stream flow graph, while attemting some quality of requirement, for example latency. From the viewpoint of the physical system, the objective is to maximize the utility of all available resources. An important goal is to maximize the number of concurrent users. Similar to a web service, a senorweb service should be impartial to all users and orchestrate its operation that benefits all users to the best of its ability. Finally, reliability has to be built into the system as a fundamental requirement.

The two major components of Sytoplasm are the *Catalog Manager* and the *Delivery Manager*. The catalog manager has a *registration* module that assigns names to regeistered streams such that they can be uniquely resolved in the global namespace. The *naming* module resolves names upon receiving requests from consumers. Finally, a *refresh* module ensures that information in the catalog is upto date to reflect the constant joining and leaving of streams. This is of particular importance to our system to allow reuse of composite streams. The delivery manager has a *planner*, a *session manager*, a *concurrency manager* and a *load balancer*. Once Sytoplasm identifies the streams required by the application, the planner generates a feed dissemination tree. At any given tree, multiple dissemination trees will be operational. We refer to each of these as a session. The session manager ensures that the content corresponding to the correct session is delivered. The concurrency manager handles concurrent access of shared streams by multiple sessions. Finally, the load balancer periodically optimizes the delivery plan to optimize resource usage.

Current Status: Currently, we have implemented only the planner and the session manager which we describe in detail next. We leave the rest as part of our future work. In addition, some other important features, for example security and privacy, is also left as future work.

4.2 Implementation

Our feed delivery system is inspired partly by geographic features inherent in sensornets and partly by network encoding. Specifically, the system builds an application-level *geographic* spanning tree for content delivery. Bottlenecks at intermediate nodes are handled by network coding.

Sensornets are inherently location-aware. A stream in itself, for example a sequence of images, is meaningless without an associated location. As such, we assume at least a fraction of all nodes to have a

notion of location. Our approach is independent of the specific protocol that assigns locations to nodes. Each nodes maintains a geographic neighbor table. The neighbor table is built by exchanging location information among peers within a predefined convex area around the node. We currently assume nodes to have fixed locations. Hence the table is updated only when a node joins or disappears.

For a consumer C, let $\{P\}$ be the set of producers that C wants content from. In order to distribute content, a tree T needs to be built that spans C and all nodes in $\{P\}$. The root of the tree is C. For each node N in T, let *Parent*(N) denote the parent of N in T. Associated with each N is a set $\{P_N\}$. In T, N is reponsible for delivering content from all elements in $\{P_N\}$ to *Parent*(N). At C, $\{P_N\} = \{P\}$. For each producer p in $\{P_N\}$, N chooses its neighbor N' that is *geographically* closest to p. N' is now responsible for delivering p to N where, N = *Parent*(N'). Thus, N partitions $\{P_N\}$ in k subsets such that,

$$\begin{aligned} \{P_N\} &= \bigcup\{P_N^i\}, \qquad i \in 1 \ldots k \\ where, N &= Parent(N^i) \\ \varnothing &= \{P_N^i\} \cup \{P_N^j\} \end{aligned}$$

This tree, called the *geographic spanning tree* (GST), is an overlay tree that is built at the application level. The GST is a reverse multicasting tree in the sense that data flows inward from the leaves to the root. The GST merely serves to provide the *next hop* information to the network layer. The network layer independently chooses a routing path between nodes. As a result, the GST might end up being inferior than a spanning tree that can be formed at the network layer. There is an obvious tradeoff between flexibility and performance. We consciously favor the former. In the next section, we will take a look at how these two compare.

At a steady state, multiple GSTs will be in operation with data from possibly overlapping producers serving multiple consumers simultaneously. As a result, data from S sources will have to be moved to K sinks. This could lead to a congestion over a particular link making routing to achieve maximum information flow impossible [21]. The problem is exacerbated by the fact that the GSTs are formed at the application level which might place a heavy load on links between a few strategically placed loads. We alleviate this problem by using two approaches: rate multiplexing and network coding.

Network coding is coding is a technique that allows mixing of data at intermediate network nodes. A

receiver sees these data packets and deduces from them the messages that were originally intended for that data sink. It was proven in [21] that this allows a sender to reach its broadcast capacity. Subsequently, it was proven that it is sufficient for the encoding functions to be linear [22]. In our case, a parent node N can encode the packets from all its children into a single packet and send it to *Parent*(N). We employ a practical coding scheme proposed in [23]. Let a GST be represented by the acyclic graph (V, E), a sender $s \in V$, and a set of receivers $T \subseteq E$. Each edge e in E emanating from a node v = *in(e)* carries a symbol *y(e)* that is a linear combination of the symbols *y(e')* on the edges *e'* entering v, namely, *y(e)* = $\sum_{e':out(e')=v} m_e(e')y(e')$. The *local encoding vector* m(e) = $[m_e(e')]_{e':out(e')=v}$ represents the encoding function at node v along edge e. For h source symbols, *y(e)* on any edge $e \in E$ is a linear combination *y(e)* = $\sum_{i=1}^{h} g_i(e)x_i$ of the source symbols, where the h dimensional vector of coefficients g(e) = [g_1(e), ..., g_h(e)] can be determined recursively by g(e) = $\sum_{e':out(e')=v} m_e(e')g(e')$. The vector g(e) is known as the global encoding vector along edge e. Any receiver t receiving along its h (or more) incoming edges e_1,...,e_h the symbols

$$\begin{bmatrix} y(e_1) \\ \vdots \\ y(e_h) \end{bmatrix} = \begin{bmatrix} g_1(e_1) & \dots & g_h(e_1) \\ \vdots & \ddots & \vdots \\ g_1(e_h) & \dots & g_h(e_h) \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_h \end{bmatrix} = G_t \begin{bmatrix} x_1 \\ \vdots \\ x_h \end{bmatrix}$$

can recover the source symbols x_1,...,x_h as long as the matrix G_t of global encoding vectors g(e_1),...g(e_h) has rank h. The interested reader is refered to [23] for further details. The information is encoded using a field size of 2^8. In each packet flowing along edge e. we include the h-dimensional global encoding vector g(e). A receiver recovers the source vectors using Gaussian elimination (details omitted). Sytoplasm also includes the timestamp in each packet. This is to avoid tuples from different instances of time to be mixed together.

Finally, a given node might be part of multiple GSTs. Data along all GSTs arrives at different rates depending upon the rate of content generation at downstream nodes. A node exploits this rate mismatch and opporunistically routes packets corresponding to different GSTs. This allows a further further efficient use of bandwidth.

4.3 Evaluation

We have implemented a preliminary version of Sytoplasm. Our prototype is implemented as a multithreaded C program where nodes communicate through sockets. We randomly deploy upto 125 nodes in a square area of side 2000m. We chose 30% of total nodes as consumer nodes. For each consumer node, we chose 35% of nodes as producer nodes. We routed 10,000 packets from each producer to its corresponding consumer. It is possible for one producer to serve more than one consumer. We mark a fraction of the nodes as geo nodes. A geo node is a node that is aware of its geographic location. We experimented by varying the number of geo nodes from 5% to 20% of total nodes. The nodes were selected at random. We assign weights to links which we take as representative of the delay. We do not model the network layer. Figure 2 shows the effect of geo nodes on path stretch. Path stretch is defined as the ratio between the GST and a network layer spanning tree. Overall, the path stretch is low. It can be seen that the stretch decreases with increase in the number of geo nodes. Figure 3 shows the median buffer occupancy across all nodes in the network. In general, the range is pretty wide. A wide majority of the nodes are not part of any GST and hence have a buffer size zero as is ahown in the CDF in Figure 4. This policy, thus, has to be supplemented by an appropriate load balancing policy that will make better use of under-utilized nodes. An almost identical trend is observed in the median number of trees that a node handles.

Current Status: Currently, we are working on simulating Sytoplasm using NS-2 [24]to obtain accurate network level masurements. We are implementing a prototype on our testbed [25] in the Cal-$(IT)^2$ building on our campus.

5 Conclusion

In this paper, we have presented Sytoplasm, a middleware architecture for enabling rich sensornets. Sytoplasm provides a stream-oriented programming paradigm and relieves application programmers from underlying details. We present a formal model of the system, the programming paradigm, the protocols and techniques for content delivery. We propose network coding techniques combined with rate multiplexing for efficient usage of bandwidth. As part of out current and future work, we would like to perform a thorough evaluation of our content delivery mechanism as well as design the remaining modules of Sytoplasm.

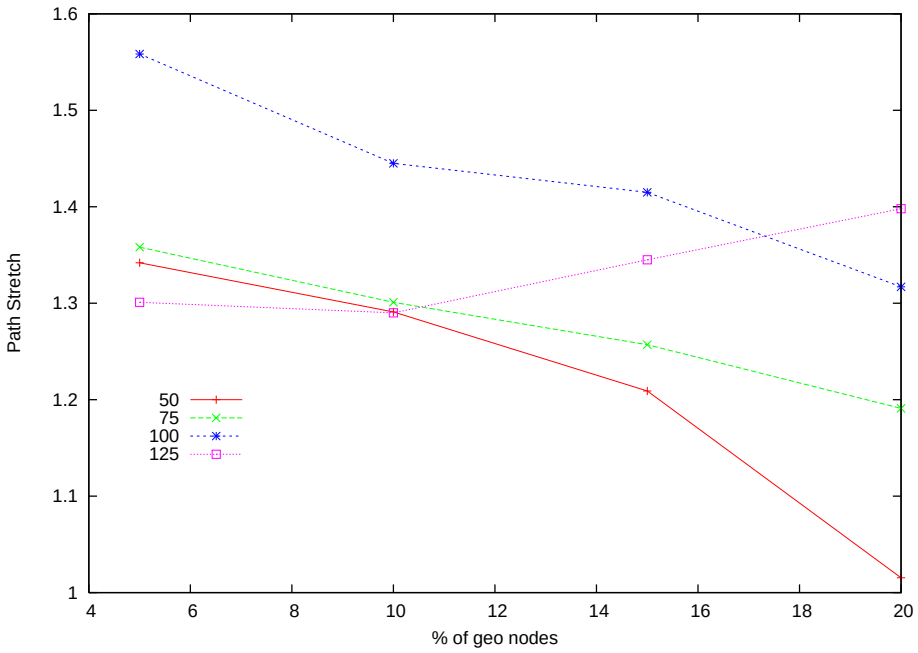

Figure 2: Path Stretch

References

[1] http://www.cisco.com/en/US/netsol/ns669/networking_solutions_solution_segment_home.html

[2] http://www.telbotics.com/pebbles.htm

[3] Zhenyu Yang, Bin Yu, Ross Diankov, Wanmin Wu and Ruzena Bajcsy, Collaborative Dancing in Tele-immersive Environment, in Proc. of ACM Multimedia (MM'06) (Short Paper), Santa Barbara, CA, 2006

[4] http://www.bittorrent.com

[5] http://www.cachelogic.com/home/pages/studies/2005_06.php

[6] V. Ramasubramanian, R. Peterson, and G. Sirer. Corona: A High Performance Publish-Subscribe System for the WorldWideWeb. In NSDI, 2006.

[7] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems. In Middleware), Nov. 2001.

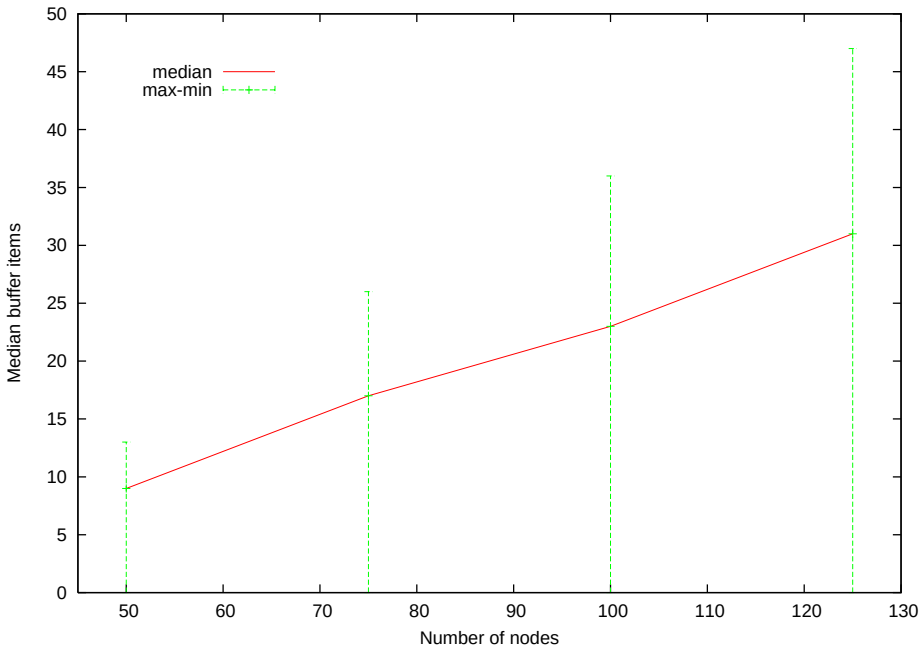

Figure 3: Median Buffer Occupancy

[8] P. B. Gibbons, B. Karp, Y. Ke, S. Nath, and S. Seshan. IrisNet: An Architecture for a World-Wide Sensor Web. IEEE Pervasive Computing, 2(4), Oct. 2003.

[9] U. Centintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. ag S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Z. k. The Design of the Borealis Stream Processing Engine. In Proc. of CIDR, Asilomar, CA, Jan. 2005.

[10] P. Pietzuch, J. Ledlie, J. Shneidman, M. R. M.Welsh, and M. Seltzer. Network-Aware Operator Placement for Stream-Processing Systems. In ICDE, Apr. 2006.

[11] http://www.rssboard.org/rss-mime-type-application.txt

[12] Cobra: Content based Filtering and Aggregation of Blogs and RSS Feeds. Ian Rose, Rohan Murty, Peter Pietzuch, Jonathan Ledlie, Mema Roussopoulos, and Matt Welsh. In Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI '07), Cambridge MA, April 2007.

[13] Js-javaspaces service specification, 2002. http://www.jini.org/nonav/standards/davis/doc/specs/html/js-spec.html.

[14] Tibco publish-subscribe, 2005. http://www.tibcom.com.

[15] Tspaces, 2005. http://www.almaden.ibm.com/cs/TSpaces/.

[16] A. Carzaniga, D. Rosenblum, and A. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. ACM Transactions on Computer Systems, 19(3):332-383, 2001.

[17] R. van Renesse, K. Birman, and W. Vogels. A Robust and Scalable Technology for Distributed Systems Monitoring, Management, and Data Mining. ACM Transactions on Computer Systems, 21(2):164 206, 2003.

[18] A. Rowstron, A. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In NGC, 2001.

[19] http://www.akamai.com

[20] C. Intanagonwiwat and R. Govindan and D. Estrin, Directed diffusion: a scalable and robust communication paradigm for sensor networks, In Mobile Computing and Networking, pp 56–67, 2000.

[21] R. W. Yeung and Z. Zhang, "Distributed Source Coding for Satellite Communications" (IEEE Transactions on Information Theory, IT-45, pp. 1111-1120. 1999).

[22] S. Y. R. Li, R. W. Yeung, and N. Cai, "Linear Network Coding," IEEE Transactions on Information Theory, vol. 49, pp. 371, 2003.

[23] P. A. Chou, Y. Wu, and K. Jain, Practical network coding, Practical network coding, 51st Allerton Conf. Communication, Control and Computing, Oct. 2003.

[24] http://www.isi.edu/nsnam/ns/

[25] http://www.ics.uci.edu/ projects/SATware/

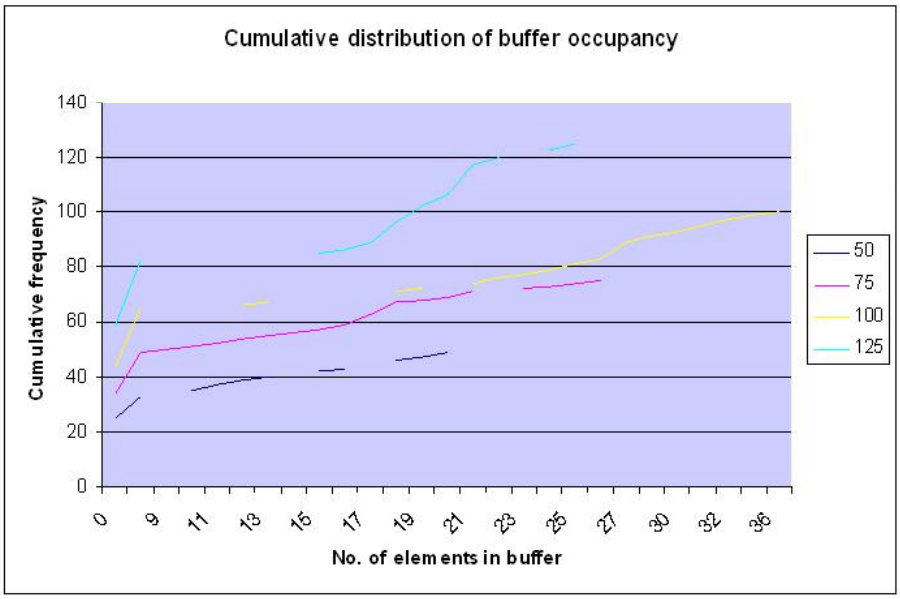

Figure 4: Cumulative frequency of buffer occupancy