

Hourglass: An Infrastructure for Connecting Sensor Networks and Applications

Jeff Shneidman, Peter Pietzuch, Jonathan Ledlie, Mema Roussopoulos, Margo Seltzer, Matt Welsh

Harvard University

{jeffsh, prp, jonathan, mema, margo, mdw}@eecs.harvard.edu

Harvard Technical Report TR-21-04

Abstract

The emergence of computationally-enabled sensors and the applications that use sensor data introduces the need for a software infrastructure designed specifically to enable the rapid development and deployment of applications that draw upon data from multiple, heterogeneous sensor networks. We present the Hourglass infrastructure, which addresses this need.

Hourglass is an Internet-based infrastructure for connecting a wide range of sensors, services, and applications in a robust fashion. In Hourglass, a stream of data elements is routed to one or more applications. These data elements are generated from sensors inside of sensor networks whose internals can be entirely hidden from participants in the Hourglass system. The Hourglass infrastructure consists of an overlay network of well-connected dedicated machines that provides service registration, discovery, and routing of data streams from sensors to client applications. In addition, Hourglass supports a set of in-network services such as filtering, aggregation, compression, and buffering stream data between source and destination. Hourglass also allows third party services to be deployed and used in the network.

In this paper, we present the Hourglass architecture and describe our test-bed and implementation. We demonstrate how our design maintains streaming data flows in the face of disconnection, allows discovery of and access to data from sensors, supports participants of widely varying capabilities (servers to PDAs), takes advantage of well-provisioned, well-connected machines, and provides separate efficient communication paths for short-lived control messages and long-lived stream-oriented data.

1 Introduction

Sensor networks present the opportunity to instrument and monitor the physical world at unprecedented scale and resolution. Deploying a large number of small, wireless sensors that can sample, process, and deliver information to external systems opens many novel application domains. Automobile navigation systems could become aware of traffic conditions, weather, and road conditions along a projected route. Buildings can be instrumented to permit firefighters and other rescuers to map ideal egress routes based on the location of fire and structural damage. Medics and physicians at the scene of a disaster can track patient vital signs and transport victims in an efficient manner based on bed and equipment availability at local hospitals.

A core challenge that emerges in this domain is that of the *infrastructure* that connects many disparate sensor networks to the applications that desire data from them. To date, work in the sensor network community has focused on collecting and aggregat-

ing data from specific networks with an associated base station. The problem of delivering this data to external systems is typically left open. At the same time, work in continuous query (CQ) systems [5, 6, 7, 16, 24] peer-to-peer overlay networks [30, 33] and publish/subscribe systems [2, 3, 4, 9, 27, 31] offers a range of techniques for similar application domains. However, these approaches do not directly address the needs of capturing and querying data from a large number of sensor networks with different data schemas, access mechanisms, and resource considerations.

This paper focuses on the problem of developing an Internet-based infrastructure that handles aspects of naming, discovery, schema management, routing, and aggregating data from potentially many geographically diverse sensor networks. We call this infrastructure a *data collection network (DCN)*. We describe the design and implementation of *Hourglass*, a DCN test-bed based on an overlay network of Internet-connected hosts that collaborate to provide a common interface for constructing pathways from sensor networks to applications. In this context, a sensor network might be a single node that collects data from a discrete physical location or a large collection of nodes capable of complex internal processing. In either case, the DCN supports data flow between applications and multiple sensor networks from different administrative domains with differing degrees of connectivity.

The essential data model in Hourglass is a *circuit*, which can be thought of as a set of network links that connect one or more sensor network sources to one or more recipients of that data. The circuit idea separates short-lived, small control messages from long-lived, stream-oriented data. Control messages are used to set up pure data channels that travel over multiple services. This is a useful idea because different types of data messages can be routed and processed using different tools.

Data flowing along a circuit may be filtered, aggregated, compressed, or temporarily buffered by a set of *services* that exist in the Hourglass infrastructure. These primitives allow applications to construct powerful ensembles of sensor networks and infrastructure services. Circuits are constructed with the aid of a *registry* that maintains information on resource availability and a *circuit manager* that assigns services and links to physical nodes in the network.

Given that sensor networks or applications receiving data from them may be mobile or connected through poor-quality

wireless channels, support for *intermittent disconnection* is a critical requirement for a DCN. Hourglass is designed to handle intermittent disconnection gracefully by instantiating *buffer services* along links in a circuit that may cross a “connectivity domain,” and by incorporating a reliability model that permits either buffering or loss of data items being routed along a circuit.

It is also the case that other nodes in the network are well-connected highly-powered nodes that offer significant computational and storage capacity. These nodes can provide complex data processing services and stable routes in the network.

The rest of this paper is organized as follows. In Section 2 we describe related work in a number of areas both from the sensor network and Internet-based distributed systems communities. Section 3 details the metropolitan-area medical data management framework that we carry through the paper as a concrete set of application requirements for Hourglass. Section 4 describes the Hourglass design in detail and Section 5 presents our test-bed implementation. In Section 6, we evaluate the use of Hourglass in a range of simulated application scenarios, showing that the DCN provides an effective means of connecting many sources of sensor data to many data recipients. Finally, Section 7 presents future work and Section 8 concludes.

2 Related Work

The application scenarios that we focus on for data collection networks rely on sensor networks, middleware, distributed query processing, and the work of many other self-contained research disciplines. However, the union of these disciplines leaves several significant research questions unanswered, because each of these fields makes a set of assumptions that do not hold for these scenarios. Here we outline related work in these areas.

2.1 Sensor networks

The field of sensor networking has received much attention in recent years as a number of research groups have constructed small computational and communication devices, such as the Berkeley motes [15], capable of interfacing with physical sensors. The research in this area has focused on how to construct useful systems while coping with the resource constraints (*e.g.*, low power, weak communication, limited processing and memory capabilities) of such devices.

Computation on the devices is used both to perform application-specific processing and also to conserve critical resources, such as communication bandwidth. For example, a large number of data samples can be analyzed *in situ*, enabling transmission of only the most critical data (*e.g.*, “this object is a tank and it has moved”) [13]. Research on specific problems includes message-passing algorithms [14, 18], on-the-fly sensor reprogramming [20], and query languages [22, 36].

The characteristic of most of this work to date is that it focuses on the wireless domain, assuming that once data has been received by a wired base station, there is a direct path to desired consumers of that data. This approach generally ignores the issues associated with constructing logical flows from multiple specialized sensor networks to distant applications; in effect they “stop at the wire.” As we will see below, these assumptions

do not hold for the large class of applications we are considering.

2.2 Continuous query processing

The Continuous Queries (CQ) work from the database community offers in-network processing of streaming data in stable, homogeneous networks [5, 6, 7, 24]. This community has addressed issues of operator placement, which is also important in a DCN. TelegraphCQ aims to work in “unpredictable” situations: nodes can fail or query optimization information can be incorrect; users can re-state their queries on the fly. However, the criteria by which CQ systems place operators do not include the possibility of intermittent connectivity. Whereas TelegraphCQ and NiagraCQ move the data to a central processing point, Hourglass can act on data either at or close to the publishing node. Additionally, these systems do not address the scalability challenge that we face, nor do they offer the wide range of data flow semantics that are necessary in our target applications.

Most closely related to the notion of a data collection network are systems such as IrisNet [8], PIER [16], and Medusa/Aurora [7], which are intended to support distributed queries over many disparate, real-time data sources using techniques such as overlay networks and dynamic query operator placement. In particular, Aurora [37] is a system designed to support applications that monitor continuous streams of data. While Aurora centralizes stream processing, Hourglass provides a distributed framework for circuit construction. PIER uses a DHT for tuple storage, spreading data around the network based on the namespace and primary key. In contrast, Hourglass creates circuits, yielding a scalable infrastructure like PIER but without the high latencies induced by PIER’s DHT architecture.

More broadly, the Hourglass approach differs from these systems in several key respects. First, we envision an extremely rich set of services that can collect, filter, aggregate, and process sensor data as it flows through a network; the DCN should not constrain the set of services to a small set of operators for a specific query interface. Such an approach allows the system to evolve to support a wide range of as-yet-unforeseen applications. Second, Hourglass is designed to cope with mobility of sensor and application endpoints and the resulting temporary disconnections from the rest of the network. Third, Hourglass dynamically incorporates heterogeneous devices into the system. CQ systems currently do not allow for this dynamic behavior, yet it will occur in long-lived applications that Hourglass aims to address.

2.3 Publish/subscribe and peer-to-peer systems

Publish/subscribe systems are one way to provide efficient content-based routing of messages to large numbers of subscribers. The publish/subscribe community has addressed issues such as congestion control [27], delivery semantics (*e.g.*, exactly-once) [2], message efficiency and battery consumption in mobile networks [38], and scalability of content-based publish/subscribe protocols [4]. In contrast, a DCN must address intermittent connectivity of both data consumers and producers and provide in-network services, like buffering, based on the stream’s semantics. Publish/subscribe inherently gives a multicast service, but one which provides little facility for QoS monitoring and control. Because publishers are a level of indirection away from subscribers, publish/subscribe does not easily permit

application-level feedback and quality of service (QoS) guarantees. In contrast, Hourglass's circuits allow for tuning QoS along a datapath.

The P2P community has focused primarily on scale and changes in system membership (churn), frequently at the expense of complex queries and a good naming system. While churn is related to disconnection in Hourglass, the goals are rather different. P2P systems typically strive to make the guarantee that when a node departs, the static data it was storing is still available. The quality of these systems is frequently expressed as the quantity of maintenance traffic required to provide these guarantees. Thus, P2P has tended to orient itself at routing around point failures of nodes and links. In contrast, our goal is to preserve the flow of data between two endpoints in the face of temporary disconnection. Even while disconnected, an Hourglass participant may be generating data for which applications are waiting, and we need to ensure that upon reconnection, the new data are correctly transmitted in the system. Thus, a central tenet of Hourglass is buffering rather than rerouting.

2.4 The Grid

The Grid initiatives have focused on naming and creating common interfaces for data and computation [10], as well as harnessing computational resources [32] and federating databases [26]. A data collection network faces similar problems; however, the Grid approach generally assumes a stable and relatively high-performance network infrastructure. In contrast, DCNs must gracefully handle temporary disconnection as well as a range of connection bandwidths to sensor networks and the application endpoints receiving sensor data. A DCN encompasses a broader and more diverse set of participants than a traditional Grid system. While common interfaces are important, the critical issue for DCNs is providing interfaces for which minimal functionality can be implemented on resource-constrained devices, which may require interfaces to be backed by sophisticated services on more capable nodes.

2.5 Research challenges for data collection networks

The goal of a *data collection network* is to allow applications to harness the functionality of many disparate sensor networks as well as infrastructure-based services such as discovery, filtering, aggregation, and storage. In many ways, the vision of a DCN is to allow Internet-based applications and services to "reach out" into the physical world of sensors that exhibit a wide variance of capabilities and connectivity. Sensor networks, especially those that are mobile and intermittently connected to the outside world, represent ephemeral entities that pose real challenges for existing Internet-based infrastructures. This paper addresses the following set of core research questions that arise in this regime.

Intermittent connectivity: How does a DCN manage communications with mobile or poorly-connected entities that may exhibit intermittent connectivity with the rest of the infrastructure? How does the DCN ensure that the data flow is not disrupted during disconnection.

Resource naming and discovery: How does a DCN infrastructure become aware of, and broker access to, a wide range of

sensor networks and services that may exist in different administrative domains, each with different interfaces and access rights?

Service composition: How do applications tie together a suite of services for processing data flowing from sensor networks? What is the model for mapping application data requirements onto individual services, and how are those services instantiated and managed? How do applications integrate application-specific processing into an existing DCN?

Supporting heterogeneity: How does a DCN infrastructure provide services in the presence of resource constrained devices such as PDAs? What minimal functionality is needed by all participants? How should it accommodate devices of varying capabilities?

As an initial step towards answering these questions, we present the Hourglass DCN infrastructure. Hourglass has the following essential features:

1. Maintains streaming data flows in the face of disconnection;
2. Provides for discovery of resources that are not generally Internet-accessible (e.g., sensor data);
3. Permits the participation of heterogeneous devices including resource-constrained devices, such as PDAs;
4. Takes advantage of the fact that some nodes are almost always connected and provide significant computational and storage capacity; and
5. Separates flow of short-lived, small control messages from that of long-lived, stream-oriented data. The circuit is the embodiment of this idea, in that control messages are used to set up pure data channels that travel over multiple services.

3 Application Framework

We describe two application scenarios we are using as our initial test-bed to provide context for the architecture that we present in Section 4. These applications derive from the world of medical and emergency sensing, and while they provide a context for discussion, it is important to bear in mind that the architecture we present is suitably general to support a much broader class of applications, such as environmental monitoring, seismic recording, large-scale and epidemiological studies.

We are concerned with improving emergency medical services in a metropolitan area, which may consist of numerous hospitals, health care facilities, ambulance companies, and 911 dispatch services.

A sensor infrastructure can incorporate wireless vital sign sensors (e.g., pulse oximetry, EKG, respiration) attached to patients; real-time sensors of traffic conditions and ambulance location through GPS; hospital and availability of beds in the emergency room; and the status of specialized facilities such as trauma and burn centers.

By allowing this information to be queried, filtered, and relayed to display terminals, handheld computers, or laptops carried in an ambulance or in the hospital, health care providers

can obtain a real-time view of the status of patients, location of ambulances, and the status of medical facilities. This vision is shared by a number of projects that are focusing on optimizing community health care for disaster response [1, 11, 19].

While there are many different applications one might construct on such a system, we describe two briefly.

3.1 In-Hospital Communication

Currently, in a hospital, critical patient events or “codes” often trigger an announcement over the hospital’s PA system, *e.g.*, “Code Blue in Ward 3.” To retain patient confidentiality, little information about the specific event is broadcast, so such broadcasts usually trigger a collection of doctors to race to the location cited. Instead, in the Hourglass-enabled hospital, sensors are attached to a patient and publish data such as vital signs about that patient into the Hourglass infrastructure. Doctors’ PDAs can register interest in the vital signs of their patients. Application-specific services in the infrastructure process this data and can trigger a direct alert to the specific doctor (or doctors) whose patient is in danger. Such an alert is transmitted directly to the physician’s PDA. When the physician leaves the hospital, her PDA continues to receive updates about the patient’s status so she can monitor the patient even as she moves to the location of her private practice.

3.2 Emergency Dispatch

The Greater Boston area is supported by sixty-eight [21, 29] independent ambulance companies and twenty-six [28] “acute care” hospitals. Currently, ambulance operators are responsible for selecting the destination to which to transfer a patient, subject to availability in the surrounding area emergency rooms. This dispatch is conducted in a somewhat ad-hoc, manually intensive manner.

In the world we envision for sensor-based emergency care, ambulances are equipped with a number of medical sensors that can be affixed to patients and that communicate with PDAs carried by the EMTs. The sensors and PDAs also communicate data to laptop-class computers in ambulances. Emergency rooms’ (ER) resource availability, *i.e.*, number of available beds, presence of particular medical specialists, etc., are managed by systems in the ER.

The Hourglass infrastructure provides a mechanism by which an Emergency Dispatch application receives data about patients in the ambulances as well as data about characteristics of Emergency Rooms. Such a Dispatch application might also receive data from non-medical organizations such as the Department of Transportation systems that provide up-to-the-second traffic reports, which can also be factored into dispatch decisions.

As EMTs from the various ambulances treat patients, the Dispatch application synthesizes data from the ambulances and ERs to direct patients to the hospital that is both geographically close and has the capabilities to provide services for the patient in question. For example, a patient suffering a heart attack would be directed by the Dispatch application to a heart catheterization lab facility, which may be further away than the nearest community hospital. Alternatively, the Dispatch application might direct the ambulances based on ER bed availability. If the near-

est ER is refusing new admissions, a different facility would be selected.

As ambulances travel to and from medical emergency sites, their wireless connections may be sporadic if they move in and out of range of wireless base stations. The Hourglass infrastructure should support mechanisms for detecting and notifying applications (such as the Dispatch application) when this happens. Moreover, the infrastructure should re-construct the flow of data from the ambulance to the interested Dispatch application when the ambulance is within range again.

With these applications in mind, we now present the Hourglass architecture and then our particular instantiation of that architecture.

4 Hourglass Architecture

In this section we describe the architecture of Hourglass. Many of the design decisions we have made in Hourglass stem from three fundamental requirements.

First, in a data collection network certain nodes are Internet-connected with reliable high speed links, whereas others are accessible only over low-bandwidth, unreliable wireless links (*e.g.*, to an ambulance). The Hourglass architecture must be able to cope with this heterogeneity of connectivity and explicitly support the disconnection of wireless nodes.

Second, data producers and some data services are completely hidden within application-specific or proprietary sensor networks. We wish to provide access to data and services in these networks, but also wish to minimize the constraints that we impose upon them.

Third, many applications, such as medical patient monitoring, require the timely delivery of data to interested parties once the data producers have been identified in the system. As a result, we separate control and data paths in the architecture and ensure that the data flow mechanism is efficient.

The rest of this section is organized as follows. We first introduce the components of the Hourglass infrastructure and explain how data is routed from sensor networks to application that have registered interest in the data through circuits. We then explain how circuits are created and disconnected. Finally, we describe how new services can be seamlessly added to the infrastructure.

4.1 Components

An example of an Hourglass system is given in Figure 1. It consists of a number of components, which we will introduce briefly here and then describe in more detail, below.

Data flow in Hourglass is based on a *circuit*, which is a data path through the system that ensures that an application receives the data in which it is interested. A circuit includes intermediate *services* that perform operations on the data. Services are organized into distinct *service providers* that capture a single administrative domain. Each service provider includes a *circuit manager*, which is responsible for the set-up and management of circuits, and a *registry*, which aids service discovery.

Hourglass services have well-specified interfaces that are used to communicate with other services, the circuit manager, and the registry for circuit establishment, data routing, circuit disconnection, and service discovery. An existing entity can join

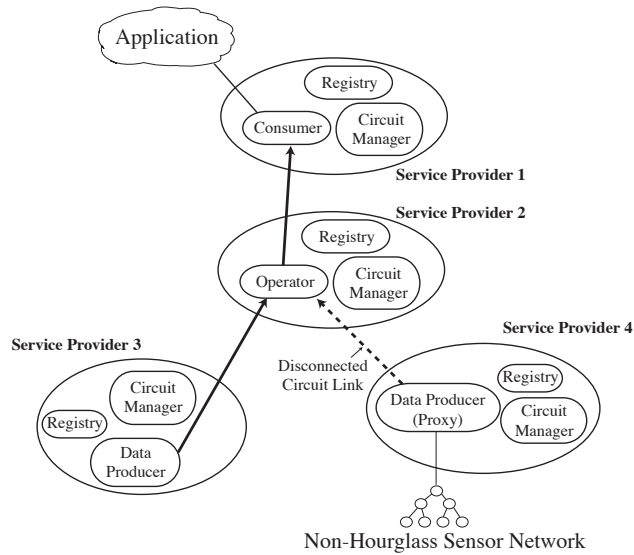


Figure 1: Example of an Hourglass system with one realized circuit. A circuit can be described by a set of circuit links between service providers (SPs) and the schema of data traveling over these links.

an Hourglass system by implementing the core functionality required by these interfaces. Sensor networks and applications may decide to attach to the Hourglass system through proxy services in order to avoid the cost of running an Hourglass service natively.

4.1.1 Circuit

A *circuit* is a fundamental abstraction that links a set of data producers, a data consumer, and in-network services into a data flow. A circuit enables applications to express their data needs at a high level and pass the responsibility for creating data flows to the DCN, thus simplifying the implementation of sensor data applications. Data injected into the circuit by data producers is processed by intermediate services and then delivered to data consumers.

As illustrated in Figure 1, a circuit in Hourglass is a tree with a data consumer as the root, and data producers as leaves. Data flows towards the consumer of the circuit and is processed at intermediate nodes. Nodes in the circuit can refer to Hourglass services in the system by including a *service endpoint* that binds a given circuit node to an actual instance of a service. A service endpoint could be implemented as an IP address and port number. Multiple circuits can share particular physical realizations of the circuit links within a circuit, avoiding duplicate transmission of data that is used by more than one circuit. A circuit also has a globally unique *circuit identifier* that is used to refer to it throughout the system.

Circuits are established by the *circuit manager* according to requests from applications. An established circuit is associated with a lease and needs to be refreshed periodically, otherwise it is removed from the system. Such a soft-state approach prevents the build-up of stale circuit information after application failures.

Note that, due to the heterogeneity of the environment, Hour-

```

<circuit>
  <consumer endpoint="192.168.0.1:16800">
    <operator topic="BostonEMSDispatch">
      <outschema v="bems:dispatchDecision1.0"/>
      <inschema0 v="bems:hospitalAvail1.0"/>
      <producer topic="BostonHospAvail"
        endpoint="192.168.15.4:16800"/>
      </producer>
      <predicate company="HMSAmbulance"/>
    </operator>
  </consumer>
</circuit>

```

Figure 2: Example of a partially-realized circuit definition in the Hourglass Circuit Description Language (HCDL). A consumer at address 192.168.0.1:16800 wants data from a specific hospital endpoint at 192.168.15.4:16800, after it has been processed by any service registered on the BostonEMSDispatch topic that matches the predicate company="HMSAmbulance".

glass does not enforce a global data model for all circuits. Instead, a single circuit can combine different data models, such as partially-structured or relational data, with a range of data schemas, as long as the services involved are able to understand each other, for example, by translating between data representations.

Circuit Definition. The structure of a circuit is specified in the *Hourglass Circuit Descriptor Language (HCDL)*. The HCDL is an XML-defined language that applications use to define desired circuits to be established by Hourglass. An HCDL circuit description can exist in three forms: the nodes of an *unrealized circuit* are not tied to actual service instances in the system, a *partially-realized circuit* contains some of the bindings, whereas a *fully-realized circuit* includes all of the service endpoints for the services used by the circuit. An unrealized circuit includes constraints on the services that may be instantiated at a certain

node in the circuit. It is the task of the circuit manager to resolve an unrealized circuit into a realized one subject to the constraints imposed by the application. These constraints come in the form of *topic* and *predicate* statements, which will be discussed in Section 4.1.2.

An example of a partially-realized circuit defined in the HCDL as created by a data-consuming application is given in Figure 2. The example shows a circuit between a hospital providing information about ER availability and a consumer interested in the dispatch decisions concerning that hospital. There are three components in the circuit: a consumer endpoint, the `BostonEMSDispatch` service, and a producer endpoint. The consumer endpoint has been instantiated at IP address 192.168.0.1:16800. The dispatch service takes its input from a data producer (hospital) that has been instantiated at IP address 192.168.15.4:16800. The input and output schema for the dispatch service have been defined, but the service itself has not been realized.

Circuit Disconnection. Often applications benefit from not having to deal with temporary disconnection of distributed components explicitly. In Hourglass, a circuit can hide the fact that some circuit links are temporarily disconnected from the data consumer. During disconnection, data can still flow through the remaining parts in the circuit. For example, when a data producer in an ambulance leaves the coverage area of the wireless network, interested parties in hospitals will still receive patient data from the remaining ambulances. However, depending on application semantics, a data consumer may register to be informed about the disconnection of data producers in its circuit, which can then result in the failure or reconfiguration of the circuit.

The disconnection of circuit links is monitored by a *heartbeat infrastructure* because TCP connection timeouts are too coarse-grained for timely detection of disconnection. Heartbeat messages are exchanged between the nodes on a circuit link to detect loss of communication between services. The heartbeat interval dynamically adapts to the disconnection behavior of a circuit link. Data messages are treated as implicit heartbeats to reduce unnecessary load on the circuit.

4.1.2 Service

The nodes in a circuit are realized by Hourglass *services*. A service can function as a pure *data producer*, a pure *data consumer*, or both. Services that implement both the producer and consumer role in a circuit are called *operators*. In the emergency dispatch scenario introduced in Section 3, the ambulances and the hospital availability services are data producers, whereas the doctors in the hospital are data consumers. The ambulance dispatch service is an operator because it consumes data that comes from the ambulances and the availability services and produces data delivered to doctors.

A service must implement two functions to produce and consume data to and from a circuit. A `produce` call takes a new data item and inserts it into the circuit. A `consume` call-back is invoked when a new data item was received by the service. This interface makes the implementation of new services simple because the complexity of having many circuits attached to a ser-

vice is hidden. The multiplexing of data to and from connected circuits is handled entirely by an Hourglass service layer.

If the data entering the Hourglass system is coming from a sensor network, individual sensors will not have enough resources to implement an Hourglass data producer. In this case, they interface with the rest of the data collection network through a *data producer proxy*. The proxy functions as the entry point for data from the wireless sensor network and implements a private communication protocol with the sensor infrastructure. In Figure 1, Service Provider 4, which could be a patient ward with an attached wireless sensor network that produces vital sign data, uses a data producer proxy to connect to Hourglass. In contrast, the data producer in Service Provider 3 includes a native implementation of an Hourglass service because it is not resource constrained.

The services in Hourglass range between *generic* services that are useful to a wide-range of applications and *application-specific* services, which are only meaningful to a single application. Generic services may or may not be tied to a particular data model. Examples for generic services are a *buffer service*, a *filter service* for XML data, and a *persistent storage service*. An ambulance dispatch service or a service to process EKG data to identify a cardiac event are application-specific services. Next we highlight the features of some generic services in more detail.

Generic Services. A *buffer service* is responsible for buffering data during disconnection and delivering it to the rest of the circuit after reconnection. It ensures that data sent over disconnected circuit links is not lost. To avoid data loss until disconnection has been detected by the heartbeat infrastructure and reported back to the application, the buffer service keeps a running buffer of data items that were sent through the circuit. When a new circuit is created whose semantics require that it be resilient in the face of disconnection, buffer services are inserted at wireless circuit links that are prone to disconnection by the circuit manager.

A *filter service* restricts the data that flows through a circuit according to a filter expression. This service depends on the data model used in the circuit. For example, a filter service that processes XML data can use an XPath expression [35] for filtering. To reduce the bandwidth consumption of a circuit, filter services should be located close to data producers. The circuit manager can relocate filter services in order to optimize the efficiency of a circuit.

Another generic service is a *persistent storage service* that enables applications to keep a history of the data that flowed through a circuit. It supports a data producer interface so that consumers can replay past data along new circuits.

Topics and Predicates. *Topics* are a convenient way to partition the space of available services in a data collection network. A topic is a “mutually-agreed-upon” way to describe semantically-related services. It is “mutually-agreed-upon” by the entities wishing to publish the existence of, and subscribe to, some service. Circuits can reference a topic in lieu of a service endpoint, and the circuit manager and registry within Hourglass will find a service endpoint that realizes this topic request.

The agreement to use a particular topic is not handled by Hourglass; the idea is that sets of users of the system will agree on appropriate topic names.

One previous system with a similar topic notion restricted the topics to be geographically-based [8]. Such an approach makes sense when services are tied to geographic locations and when a system is used for one type of service, neither of which apply to Hourglass.

Requiring this out-of-band agreement is not unusual. Consider the telephone book as an analogy: looking up a name in the yellow pages works even in the absence of explicit mutual agreement. If one is looking for a wedding singer, one might look under topics “Entertainment” or “Wedding Services”.

Note, however, that many different services can be found in “Wedding Services”. Since many services may exist on a particular topic, finding a specific service is aided through the use of *predicates*. Predicates are logical statements asserting some service property, or feature of the data generated by this service. For instance, in the telephone book example, a predicate could take the form of `dj.music="80s"`.

When a service announces itself to the Hourglass system, it affiliates with one or more topics and further describes itself using a set of predicates. When a partially- or unrealized HCDL circuit descriptor is given to the circuit manager, it can contain predicates that will be used to match services. This evaluation is performed by the registry in response to a message from the circuit manager.

In both the telephone book and in Hourglass, more than one service may match the set of predicates and topics. In this case, it is up to the circuit manager to decide which service to use. A circuit manager could make this decision randomly or more intelligently, perhaps by load-balancing or picking geographically close servers. (Quality of Service information is returned via the registry.)

4.1.3 Service Provider

The services in Hourglass are arranged into *service providers* (SPs). A service provider is comprised of one or more Hourglass *nodes*. Each service provider is contained in a single administrative domain and an SP enters and leaves the Hourglass system as a unit. An SP must support a minimum functionality in the form of a circuit manager and a registry to join an Hourglass system. Even though the minimum functionality required by a service provider is small, a set of additional services is suggested. In particular, SPs that wish to retain data across disconnection must provide a buffer service. A filter service enables data consumers to restrict the data flow at the source, thus potentially saving wireless bandwidth.

In general, service providers can be heterogeneous in size. In the medical domain, an ambulance and the sensors associated with it would comprise a single service provider that disconnects when the ambulance goes out of range. A hospital or a dispatch service might act as another service provider. In an ambulatory monitoring application, the PDA carried by a patient could act as a lightweight service provider that includes a data producer of medical data. Four service providers are shown in Figure 1.

Some service providers are relatively stable in that they are well-connected and rarely leave the system. Other SPs are less

stable and are expected to become disconnected. A service provider is associated with a *maximum disconnection interval*. Service providers with an interval of zero form the stable core SPs of the Hourglass system and will only involuntarily disconnect due to failure. Recalling our target applications, hospitals and dispatch services are examples of stable SPs, while laptops in ambulances are examples of less stable SPs, as they are expected to periodically move out of connectivity range. When a service provider is disconnected from the rest of the system, circuits among its local services can still be formed using the local circuit manager and registry.

4.1.4 Circuit Manager

The circuit manager (CM) takes a circuit request in HCDL (see Figure 2) as input and manages the circuit creation process. The CM must take or create a fully-realized circuit descriptor and instruct the appropriate service endpoints to form links to each other, possibly with some implementation-specific recovery mechanism if an error occurs. Once all links are formed, the CM signals services to start data creation and processing.

In more detail, the CM’s job starts when given an HCDL document. If the HCDL descriptor is not fully-realized, meaning that at least one of the operators in the circuit has been described abstractly rather than tied to a particular implementation point, the CM must send a lookup request to the registry to get endpoint addresses for these operators. As discussed in Section 4.1.2, a list of service matches is returned to the CM, and one matching service is picked. (If no match is found, the circuit creation fails.)

The CM is capable of re-structuring a circuit request to better fit current system usage. For instance, the circuit manager may choose to insert a buffer service when a link appears to disconnect frequently and an application wishes not to lose data. (In the future, these loss tolerances could be expressed in the HCDL, though an implementation of the CM could use an alternate algorithm to decide what it thinks an application can tolerate.) An intelligent decision by a capable CM is possible since much of a service’s QoS information, such as load and connectivity information is known to that service, and this information is refreshed at some service-specific frequency into the registry.

Every service provider must have a circuit manager so that if an SP gets disconnected, it and other CMs could each act to recover from a potential inconsistencies in state due to the disconnection.

Finally, a circuit manager is responsible for inserting the running circuit details into the registry under a special control topic so that other circuit managers can find and optimize this circuit. Intelligent circuit optimization is a subject of future work.

4.1.5 Registry

The registry is a distributed repository of information about the various services and circuits. It is best described as a distributed lookup service, and could be implemented in a number of ways with a number of technologies [17, 23, 30, 33].

Service information is stored in the registry so that applications can locate interesting producer endpoints, and so that the circuit manager can realize portions of a circuit. Existing circuits

are stored in the registry to allow circuit managers (optionally) to optimize already running services.

Every service provider must have a registry. In addition to the disconnection tolerance, this allows connected registries to share the registry management tasks. For instance, if the registry backing scheme is a distributed hash table, one can imagine the registries coordinating to find the more stable members, and then dividing topic ownership among these members.

In addition to performing topic management (which can include splitting the load, replication, etc.), the registry must provide predicate filtering. As described in Section 4.1.2, a circuit descriptor can constrain an unrealized entry with these logic statements. When the circuit manager attempts to realize a service, it forwards these predicates to the registry, which then must perform a matching algorithm. An advanced registry could allow interesting predicate operators over a wide variety of data types, whereas a minimal registry is required to support equality on string and integer data types.

Because of the problem of node disconnection, and because it should be possible for connected subsets of services to continue operating even after a disconnection, a registry must always store information for *local* services, in addition to whatever *global* responsibilities it may have.

4.2 Service Announcement

When a service is activated within a service provider, it must declare its existence to the local registry. It does this with an *announce message*, which must contain a communication endpoint, topic name, and a lease time declaring how long it should stay registered in the system. (We use leases as a way of cleaning up in case of long disconnection.)

In addition, a service announcement can contain predicates that describe that service. These predicates should hold true for the duration of the lease, and a service should try not to violate these predicates when publishing data. (For instance, a data producer backed by a sensor might declare itself always to publish data from Cambridge. If a sensor is moved out of Cambridge while the lease is still active, the stored registry information will be incorrect.) For this reason, it is often important that an application employs a filter service in their circuit to ensure that only correct data is propagated, and it is possible that a registry entry will be out of date.

Finally, the announcement can contain quality of service information, such as the service's current load, uptime, etc. This can be used by intelligent circuit managers to perform optimization.

Once the service has announced itself to the registry, it must periodically renew its lease with the registry, but is otherwise ready to receive circuit connections.

4.3 Circuit Establishment

The establishment of a new circuit is initiated by an application. To interact with Hourglass, the application has to implement an Hourglass service. This happens either directly if the application is hosted on a machine with sufficient resources, or indirectly by using a proxy service that manages the Hourglass interaction on behalf of the application. The protocol for circuit establishment is illustrated in Figure 3.

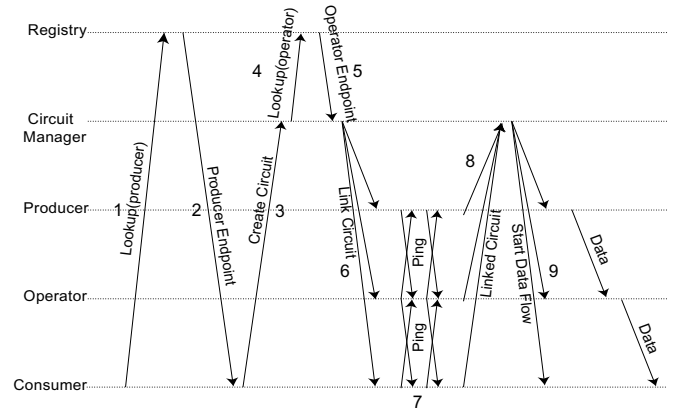


Figure 3: Process of Circuit Establishment

The data consumer that wishes to establish a new circuit first creates an unrealized HCDL description of the desired circuit. Then, the data consumer must retrieve a suitable set of data producers for the circuit from the registry (step 1). The data consumer passes a query to the registry and receives a result set of matching data producers back (step 2). In case this set is too large, the query can be refined iteratively with stricter matching predicates. The consumer may also decide to realize some of the operators in the circuit to particular instances of Hourglass services in the system. The data consumer now updates the circuit definition to include the data producers and contacts the circuit manager with a circuit creation request that includes the partially-realized circuit definition in step 3.

It is the responsibility of the circuit manager to realize the remaining services without service endpoints in the circuit. It does this through lookups to the registry (steps 4–5). After the circuit description is fully-realized, the circuit manager contacts the involved services (step 6). The services receive relevant subsets of the full circuit description and form circuit links to their parent and children (step 7). After that, the services report the successful creation of circuit links back to the circuit manager in step 8. After verifying that all circuit links for the circuit were established, the circuit manager initiates the flow of data in that circuit (step 9).

During the lifetime of the circuit, the data consumer has to periodically refresh the circuit with the circuit manager to prevent the expiration of its lease. Note that the application is shielded from the complexity of dealing with the circuit set-up by interacting with the circuit manager instead of individual services.

4.4 Data Routing

Data items are routed from producers to consumers along the paths created by circuits. A data item is associated with *list of circuit identifiers* of the circuits that it follows. This connection-based routing approach makes the routing decision at each node trivial, resulting in a low latency overhead for data dissemination in Hourglass.

By default, a produced data item will follow all connected

circuits of a service. However, an Hourglass service has the option of exercising control over the association of data items with circuits by explicitly stating the list of recipient circuits identifiers. The multiplexing of data items to circuits is done by the Hourglass service layer. The service layer reuses physical connections between services as much as possible. It also ensures that a data item, which is associated with multiple circuits connected to the same service endpoint, is only sent once through a physical network link.

4.5 Circuit Disconnection

When a service provider is disconnected from the rest of the Hourglass system, local services notice that the lack of heartbeat messages on their circuit links. A service may initiate an application-specific action, such as buffering data on the circuit, when it detects disconnection. Upon reconnection, the normal flow of data through the circuit can resume.

When the circuit manager loses connectivity to outside components, it prevents local services from setting up circuits involving unavailable service providers. The creation of local circuits within the SP is unaffected by disconnection, so that, for example, a disconnected ambulance can still disseminate sensor data from a local EKG to a monitoring station within the ambulance. This means that a disconnected registry is also able to provide information about local services and circuits and supports the announcement of new local services. The local registry reconciles its state with the global registry when the SP rejoins the Hourglass network, as mentioned in Section 4.1.5.

5 Implementation and Test-Bed Environment

In this section, we describe the implementation of the Hourglass system that we have developed and demonstrate how the design decisions we have made address our target application characteristics. The implementation is written in approximately 10,000 lines of Java. It consists of a circuit manager, a registry, an abstract service package that facilitates the implementation of new services, and several concrete services, such as a buffer service, a filter service, and various data producers and consumers. We implemented our Hourglass prototype on top of ModelNet [34], an emulation environment for large-scale distributed systems. This enabled us to set up experiments with a significant number of machines on a non-trivial network topology. It will also lead to simple deployment in the future, as all communication already travels over standardized channels. Next, we will describe the implementation of the main Hourglass components.

The implementation of an Hourglass service has three layers,

1. an *application layer* that contains application-specific data processing,
2. a *service layer* that implements generic Hourglass service functionality, and
3. a *messaging layer* that is responsible for communication via TCP connections.

The bottom two layers are provided by the abstract service package and are reused between service implementations. Since our implementation is constrained by the capabilities of ModelNet and our physical hardware, our goal was to keep the number

of required threads and open TCP connections as low as possible. For this, we use asynchronous I/O for the messaging layer, enabling us to handle a large number of connections with few threads. The messaging layer also maintains a pool of open TCP connections to other services, circuit managers, and registries that can be reused on demand. Heartbeat messages verify the liveness of these connections and generate an up-call into the service layer when disconnection or failure has occurred.

For simplicity of implementation, our circuit manager and registry are also implemented as Hourglass services, which understand a richer set of control messages. Our circuit manager and registry are currently quite simple. Our CM, for instance, does not perform any optimization, and must be given explicit references to buffer services.

The registry is implemented using hashes of topic names. On receiving an announcement, a registry adds the entry into its local store and forwards the request to the node that is currently the root of the hash. Because nodes in the core are assumed to exhibit low churn rate, we opted for low-latency one-hop lookups [12]. Disconnected nodes not in the core forward announcements when they become reconnected to the core and they are not used as roots of topics. No sophisticated indexing mechanisms are employed by the registry implementation for predicate matching of requests in the current system.

ModelNet gives the illusion of many network *client* interfaces connected to a *network core*. In our topologies, there are 50 virtual routers in the core that can be configured to provide different link characteristics such as latency, loss, and bandwidth limitations. In addition, we were able to use ModelNet tools to inject faults into the network in real-time during our experiments. Thus, only through heartbeat messages were nodes able to discover that links had failed, as would be the case in deployment.

The core is run on a network of FreeBSD machines, to which a number of Linux client machines are attached. Our environment contained eight IBM Blades each with two hyper-threaded processors and four GB of RAM. All of the machines are connected via a 1 gigabit switch. While each client is capable of supporting many network interfaces and running many clients, we ensured that all traffic between different address endpoints running on the same physical host was routed through the core. We also ensured that the physical network and core machines were never saturated during our experiments, which would have led to an artificial loss of packets.

6 Evaluation

We evaluate our system on two levels: The first, most basic criteria is that our test-bed performs well enough to support interesting experiments by us (and by future researchers), and that our implementation algorithms scale well to reasonably large experiments. As described in Section 3, our first implementation targets a metropolitan medical scenario. We expect that the number of concurrent circuits to be in the thousands, the number of services to be in the hundreds, and the number of services in use by each circuit to be in the low single digits.

The second, and more interesting evaluation, has to do with functionality of our data collection network implementation. We address this by implementing a sophisticated scenario from the

Connection Type	Max. Supported Circuits
GPRS	12
Wireless 802.11b	2424
100-base-T LAN	32328

Figure 4: Maximum number of circuits that are supportable given a connection type. This assumes an 80% link utilization and that heartbeats are exchanged once every three seconds.

medical domain, and evaluate its operation under disconnection.

6.1 Scalability

Our goal in evaluating performance is a sanity check to run large experiments. Our experimental methodology was to set up a series of experiments in our test-bed running on the ModelNet emulation, and verify that our implementation scaled as we increased the size of the experiment. The absolute numbers given in this section are meant to convey the overhead of our current test-bed, which primarily comes from Java object serialization, which we used for simplicity in development. As we will show, the encoding overhead is acceptable.

Circuit setup and overhead is verifiably light. Figure 5 shows the process of circuit creation, instrumented at points to show elapsed time. For a minimal circuit creation, in which a consumer knows and connects to a single producer endpoint, a consumer sends a 4490-byte request to the circuit manager.

The circuit manager sends a *linkCircuit* messages to each participant in the circuit. The minimal *linkCircuit* request is 1720-bytes. A producer responds to the *linkCircuit* with a 3290-byte *linkedCircuit* response. Once the circuit has been established, the circuit manager sends a 3412-byte message to start the circuit, and data flows from the circuit to a consumer, perhaps via one or more operators.

While the circuit is active, each set of endpoints on a circuit link exchange 464-byte heartbeat messages. The heartbeat period is a tuneable parameter, but is on the order of several seconds. Table 4 shows the maximum number of circuits that are supportable given a connection type, including TCP overhead, conservatively assuming that each circuit is disjoint.

6.2 Implementation Functionality

The previous section demonstrated that the Hourglass protocol is rather lightweight both in terms of elapsed time and bandwidth consumed. In this section, we focus on the unique properties of Hourglass: maintaining data flow in the presence of disconnection and the ability to place services arbitrarily within circuits.

The implementation of a new Hourglass service is simple as it only involves the implementation of a production and consumption interface. For the following experiments, we implemented an ambulance dispatch scenario in our test-bed, in which ambulances are connected via wireless links to an ambulance dispatch service. The ambulance dispatch service receives vital sign data from patients in ambulances and makes a dispatch decision for the ambulance depending on hospital availability information, coming from a number of hospitals. The patient data is then sent to the allocated hospital, while the ambulance is in transit. Note that it is important that ambulances are dispatched and patient

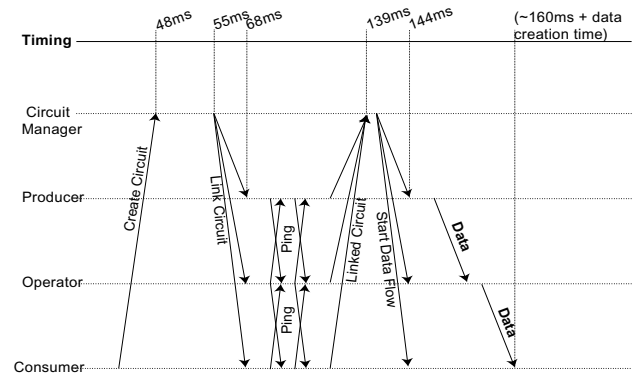


Figure 5: This figure shows the logical structure of a simple circuit creation including the latency breakdown for each component of the process. Numbers are an average over 500 circuit creations, with a low data variance. An application creates a circuit by contacting the circuit manager. The circuit manager creates links between each set of adjacent services in the circuit. (Any unknown circuit nodes are realized with the registry, an optional step that is not shown. If required, the registry request/response adds a computation delay that is linear in the number of services registered in this registry and a communication delay that is linear in the number of matching services.) The circuit components, once coordinated with each other, report their ready status to the circuit manager, which then triggers data flow through the circuit.

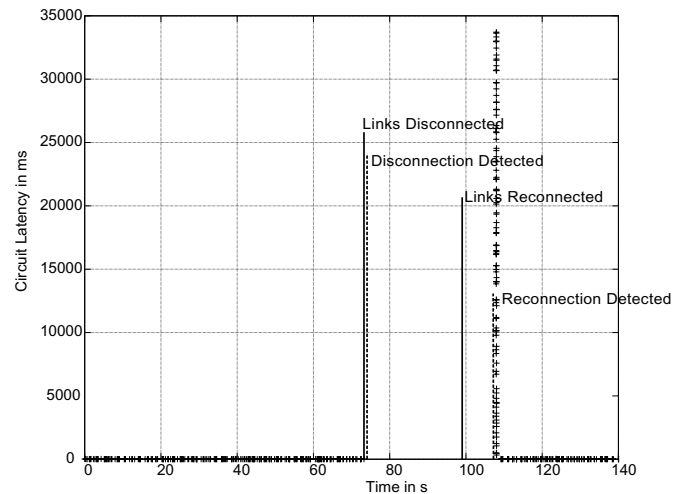


Figure 6: In this microbenchmark experiment, 2 circuits are created by doctors in two different hospitals. After around 72 seconds into the experiment, all links to ambulances are taken down. This is detected by the buffer service, which stores ambulance patient data until the links are restored at around 108 seconds. Whereas disconnection detection occurs quickly due to our heartbeat protocol, the reconnection is dependent on TCP's reconnection strategy, which is part of the TCP/IP stack implementation.

data delivered even when some ambulances are disconnected from the system. Buffer services in the ambulances ensure that no data is lost during periods of disconnection. The circuits in this scenario are set up by doctors on duty in the hospital. All es-

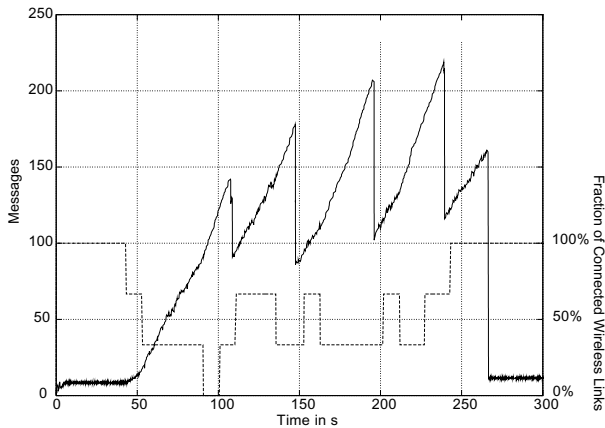


Figure 7: This experiment demonstrates disconnection behavior on a slightly larger scale than that of Figure 6. This system consists of 6 circuits created by doctors to 3 ambulances. Like the previous figure, the circuit manager has inserted buffer services running in each of the ambulance service providers. Unlike the previous figure, here ambulances are brought in and out of the system (shown in the lighter line read on the right-hand axis.) The darker line plotted against the left-hand axis shows the messages queued in the system as links go down and come back up. At time $t = 46s$, the first link goes down and the first buffer service starts to fill. The remaining links to other ambulances soon follow, causing the slope of the messages queued line to increase. Over time, on re-connections, the buffers dump their data to the consumer. At time $t = 242s$, all ambulances are once again in range. Because of TCP back-off, it takes the system about $25s$ to discover the final re-connection, and all data is finally dumped. buffer services

established circuits re-use the data connections to ambulances and can be expressed as concise HCDL XML documents.

7 Future Work

Our prototype system addresses only a few of the important research problems in this domain; much remains to be done. Currently, we perform little optimization during circuit management; however, this is a rich area of research. In the abstract, circuit optimization is similar to the problem of query optimization in a distributed database system. Conventional database query optimization techniques typically rely upon a great deal of global knowledge about the system, and this is neither present nor practical in our scenario. Our participants are (potentially) in separate administrative domains and are subject to changes in workload that arise from a variety of different sources (*e.g.*, CMs may be operating at a large number of SPs and a single SP providing a hot service might suddenly become overloaded). Creating and recreating circuits to provide reliable data and quality of service guarantees will be an interesting challenge.

Although we target the Hourglass architecture for global scale, our current implementation faces severe limitations. For example, having all filter services resolve to a single location in the registry is both inefficient and impractical. We expect that incorporating hierarchical SPs (SuperSPs) that can act as a single SP will address this problem and allow us to scale significantly.

Our current circuit semantics are designed for long-lived,

streaming data. However, there are applications that can benefit from short-lived message exchanges. We will try to unify these semantic differences by including efficient content-based routing algorithms for datagrams [25] in addition to circuit establishment.

As is the case in most system designs, naming also presents a new set of challenges. The current Hourglass naming scheme (*i.e.*, topic subscriptions) assumes *a priori* coordination between data producers and data consumers. Although practical for our current scale, this will break down under greater scalability and more heterogeneous use. Attacking this without getting drawn into trying to solve the full ontology and schema integration problems will require carefully selected trade-offs.

8 Conclusions

We have presented the Hourglass infrastructure, which addresses a number of challenges that must be overcome in order to enable the widespread deployment of sensor network applications. We have designed an architecture and testbed that maintain the potentially long-lived logical flow of data between producers and consumers in the face of intermittent connectivity and tremendous variation in participant capabilities. The circuit manager and registry together construct these logical data flows, providing access to data typically hidden within sensor networks, and taking advantage of the differing connectivity qualities between participants in the system. The architecture provides the ability to inject generic or application-specific services into these logical data flows efficiently and robustly. Many more open problems exist, but the Hourglass infrastructure provides a flexible framework with sufficient functionality to enable exploration of the next generation of research questions in this area.

References

- [1] MobiHealth Project IST-2001-36006, EC programme IST. <http://www.mobihealth.org>, 2002.
- [2] S. Bholra, R. Strom, S. Bagchi, Y. Zhao, and J. Auerbach. Exactly-once Delivery in a Content-based Publish-Subscribe System. In *2002 International Conference on Dependable Systems and Networks (DSN 2002)*, Bethesda, MD, June 2002.
- [3] A. Carzaniga, D. Rosenblum, and A. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions of Computer Systems*, August 2001.
- [4] R. Chand and P. Felber. A scalable protocol for content-based routing in overlay networks. Technical Report RR-03-074, Institut EURECOM, February 2003.
- [5] S. Chandrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *First Biennial Conference on Innovative Data Systems Research (CIDR 2003)*, Asilomar, CA, January 2003.
- [6] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagraCQ: A Scalable Continuous Query System for Internet Databases. In *SIGMOD / PODS 2000*, Dallas, TX, May 2000.
- [7] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable Distributed Stream Processing. In *First Biennial Conference on Innovative Data Systems Research (CIDR 2003)*, Asilomar, CA, January 2003.
- [8] A. Deshpande, S. Nath, P. Gibbons, , and S. Seshan. Cache-and-

- Query for Wide Area Sensor Databases. In *SIGMOD 2003*, San Diego, CA, June 2003.
- [9] P. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec. The many faces of publish/subscribe. In *ACM Computing Surveys (CSUR)*, 2003.
- [10] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Grid services for distributed systems integration. *IEEE Computer*, 35(6), 2002.
- [11] D. Gagliano and Y. Xiao. Mobile Telemedicine Testbed. *Proceedings of the American Medical Informatics Association (AMIA) Fall Symposium*, 1997. National Library of Medicine Project NO-1-LM-6-3541.
- [12] A. Gupta, B. Liskov, and R. Rodrigues. Efficient Routing for Peer-to-Peer Overlays. In *NSDI 2004*, San Francisco, CA, March 2004.
- [13] J. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, and D. Ganesan. Building efficient wireless sensor networks with low-level naming. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, October 2001.
- [14] W. Heinzelman, A. Chandrakasan, and H. Balakrishnan. Energy-efficient Communication Protocols for Wireless Microsensor Networks. In *Hawaii International Conference on System Sciences (HICSS)*, Maui, Hawaii, January 2000.
- [15] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *ASPLOS-IX*, November 2000.
- [16] R. Huebsch, J. Hellerstein, N. Lanham, B. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *VLDB'03*, Berlin, September 2003.
- [17] Java Naming and Directory Interface (JNDI). <http://java.sun.com/products/jndi/>.
- [18] D. Johnson, D. Maltz, and J. Broch. *DSR: The Dynamic Source Routing Protocol for Multi-Hop Wireless Ad Hoc Networks*. Addison-Wesley, 2001.
- [19] V. Jones, R. Bults, and D. Konstantas. Healthcare pans: Personal area networks for trauma care and home care. In *Fourth International Symposium on Wireless Personal Multimedia Communications (WPMC)*, September 2001.
- [20] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks. In *NSDI 2004*, San Francisco, CA, March 2004.
- [21] Licensed Region IV Ambulance Services. <http://www.mbemsc.org/region/ambsvcs.htm>.
- [22] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. The Design of an Acquisitional Query Processor for Sensor Networks. In *SIGMOD 2003*, San Diego, CA, June 2003.
- [23] P. Mockapetris. Domain Name Standard: RFC 1034, November 1987.
- [24] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query Processing, Resource Management, and Approximation in a Data Stream Management System. In *First Biennial Conference on Innovative Data Systems Research (CIDR 2003)*, Asilomar, CA, January 2003.
- [25] G. Mühl, L. Fiege, F. Gärtner, and A. Buchmann. Evaluating Advanced Routing Algorithms for Content-Based Publish/Subscribe Systems. In *IEEE MASCOTS 2002*, October 2002.
- [26] Petascale virtual-data grids. <http://www.griphyn.org/projinfo/intro/petascale.php>.
- [27] P. Pietzuch and S. Bhola. Congestion Control in a Reliable Scalable Message-Oriented Middleware. In *Middleware 2003*, Rio de Janeiro, Brazil, June 2003.
- [28] Region IV Hospitals. <http://www.mbemsc.org/region/hosp.htm>.
- [29] Region IV Map. <http://www.mbemsc.org/region/map.htm>.
- [30] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*, Heidelberg, Germany, November 2001.
- [31] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *NGC 2001*, UCL, London, November 2001.
- [32] Seti@home. <http://setiathome.ssl.berkeley.edu>.
- [33] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, August 2001.
- [34] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kotic, J. Chase, and D. Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, December 2002.
- [35] W3C. XML Path Language Version 1.0 (W3C Recommendation), November 1999.
- [36] Y. Yao and J. E. Gehrke. Query Processing in Sensor Networks. In *First Biennial Conference on Innovative Data Systems Research (CIDR 2003)*, Asilomar, CA, January 2003.
- [37] S. Zdonik, M. Stonebraker, M. Cherniack, U. Cetintemel, M. Balazinska, and H. Balakrishnan. The Aurora and Medusa Projects. In *Bulletin of the Technical Committee on Data Engineering*, March 2003.
- [38] H. Zhou and S. Singh. Content based multicast (CBM) in ad hoc networks. In *Proceedings of the 1st ACM international symposium on Mobile ad hoc networking & computing*, Boston, MA, 2000.