

iQueue: A Pervasive Data Composition Framework

Norman H. Cohen, Apratim Purakayastha, Luke Wong, Danny L. Yeh
IBM Thomas J. Watson Research Center, Hawthorne, New York, USA
{ncohen,apu,lukew,dlyeh}@us.ibm.com

Abstract

There will soon be a huge number of data sources accessible to applications across the Internet. These include web services, personal devices such as cellular phones and cars, and sensors measuring physical phenomena. New classes of data-composition applications can exploit this data. However, the data is diverse, voluminous, and often rapidly changing. The sources of the data can be mobile, distributed, and failure-prone. Without system support, applications that use this kind of data are difficult to write. The **iQueue** data composition framework provides system support for data composition, thereby making the task of writing applications easier.

1. Introduction

There is an explosion in the number of data sources accessible across the Internet. They include *millions* of files, databases, and online data feeds; *billions* of personal devices such as cellular phones, wireless PDAs, and networked cars; and *trillions* of physical data sources, such as traffic and temperature sensors. The data is voluminous, diverse, and often dynamic, with rapidly changing values. Novel responsive applications can be built by efficiently *composing* data from these diverse sources. Examples of the kinds of applications we envision are:

- Let me know when the stock value of PQR Inc. is \$14 and my friend has wired the \$1000 that he owed me.
- Let me know if DVD players are on sale in an electronics store I will be driving by soon.
- Based on current taxi locations and activities, assign the next passenger pickup to a particular taxi.
- Continuously suggest driving routes, based on data from traffic sensors and traffic data feeds.

These applications involve programmable composition of rapidly changing data, often with mobile sources.

It is burdensome for applications to manage composition by themselves. The burdens include coping with the diversity and heterogeneity of data sources, dynamically selecting appropriate data sources, and scaling composi-

tion of data from mobile, distributed data sources [1]. The *iQueue* data-composition framework allows applications to create *composers*; to specify a composer's data sources using functional *data specifications*; and to specify a composer's computation through application-specific code or a library of built-in primitives. Once a composer is activated, the iQueue run-time system selects data sources satisfying the data specifications, dynamically reselects data sources as appropriate, mediates between diverse data formats, and manages network placement of composers. The iQueue system enables applications to focus on the *semantics* of composition by facilitating the *mechanics* of composition.

1.1. A closer look at two examples

This section discusses two examples in greater detail and motivates the need for a composition framework.

1.1.1. A simple example: composing a shopping list with local ads. Consider an application to compose a mobile subscriber's location data, his shopping-list file, and data from local "yellow-pages" services to alert him to local purchasing opportunities of interest. Suppose that several localities have their own "yellow-pages" network services that post ads for local businesses. Different local yellow-pages services use different protocols, naming conventions, and ad formats. There are several possible sources for subscriber location data: Sometimes it may be appropriate to use location data from a GPS receiver in a car, and sometimes it may be appropriate to use location data available from cellular-phone carriers.

In the absence of system support, this application must determine the appropriate source for obtaining the user's location, invoke the mechanisms required to obtain data from that source, and interpret that source's format for the location data. Given a location, the application must identify a local yellow-pages service. It must invoke the mechanisms required to obtain ads from the selected yellow-pages service, and interpret that service's ad format to compare ads with shopping-list entries. Meanwhile, the application must monitor the location data to

determine when to find a new yellow-pages data source.

By managing the many possible variations in data sources, iQueue simplifies the writing of this application. The iQueue system resolves a functional data specification for the end user's current location, determining the best data source for this information. The system, possibly invoking application logic, then determines the best yellow-pages data source for the current user location. It converts ads obtained from this source to a canonical format, simplifying the writing of application code that matches ads with the shopping list. The system monitors changes in the current location, and automatically rebinds to a new yellow-pages service when appropriate.

1.1.2. An intricate example: composing traffic data.

Consider the problem of finding the fastest route between two points, based on data from hundreds of thousands of roadway traffic sensors backed up by less timely, less accurate data from a manually updated traffic-report web service. In case of an isolated sensor failure, local traffic conditions are inferred from other nearby sensors. If several sensors fail in the same area, the computation must rely on the web-service data for that area.

The amount of raw traffic-sensor data is immense, and centralized processing of the raw data will not scale. Raw data from contiguous sensors must be correlated to deduce traffic-movement patterns, and averaged over time to smooth out the effects of traffic lights. Large volumes of sensor data from an area around a given point can be reduced to a single traffic-speed measurement for transmission to a central computation. This data reduction should occur at the edge of the network, close to the traffic sensors, to minimize network congestion. Congestion can be further reduced by transmitting data only in response to an explicit request, and caching data received from certain heavily traveled, often queried locations.

The iQueue system simplifies the writing of this application by managing details of distribution. Hierarchies of composers are grown dynamically to compute paths in particular regions and subregions, and composers are deactivated after they have been idle for a while. The system automatically resolves specifications for traffic data for a given region to data inferred from sensors if possible, and to data obtained from the web service otherwise. A sensor failure is automatically detected, and rebinding initiated, when the sensor's previous service announcement expires without a new one having arrived.

1.2. Structure of the paper

Section 2 provides an overview of the iQueue system. Section 3 discusses the programming model and the associated data model. Section 4 describes the architecture of the iQueue runtime system that implements the programming model. Section 5 discusses the current

status of our implementation. Section 6 discusses related work. Section 7 concludes and outlines future work.

2. System overview

An iQueue application obtains its data from composers. A given composer combines particular kinds of data and produces a particular kind of result. A composer both reports the values that it computes, on demand, and accepts subscriptions for notifications of new values.

A composer's data sources may include both external data sources—such as files, databases, newsfeeds, Simple Object Access Protocol (SOAP) [2] web services, and sensor data—and other composers. Both kinds of data sources are described by data specifications, which are descriptions of *what kind* of data a composer needs rather than of *where* that data can be found. The iQueue system finds a source for the specified data. In some cases, there may be several possible data sources, and the most appropriate choice may depend on system conditions (such as network congestion) or real-world conditions reflected in data obtained from other data sources. These conditions may change from time to time, so iQueue supports *continual rebinding* of data specifications to the currently most appropriate data sources. The binding of a composer's data specifications to other composers results in a hierarchy of composers, such as that shown in Figure 1.

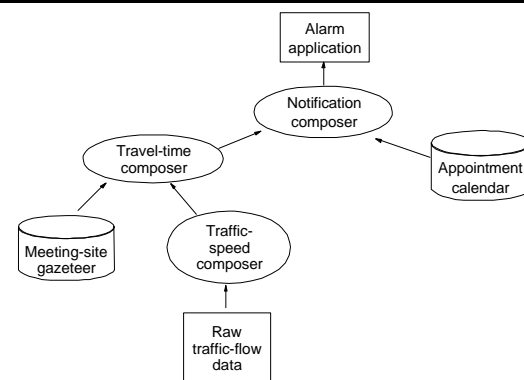


Figure 1. An iQueue application to issue an alarm to leave for an appointment, based on a user's calendar and road traffic. The application subscribes to the notification composer for notifications that it is time to leave, displaying an alarm when one arrives. The notification composer has two data sources: the calendar and a composer providing the time needed to travel to a specified location. The travel-time composer has two data sources: a gazetteer to translate appointment locations into map coordinates and a composer that computes traffic speed between pairs of map coordinates. The traffic-speed composer uses raw traffic flow data obtained over the network.

Figure 2 depicts the internal components of a single host in a distributed iQueue system. An application program invokes the *composer manager* to create a composer as an instance of a *composer specification*. As part of the process of creating the composer, the composer manager passes a data specification and an application-provided *binding module* to the *binding manager*. The binding manager passes the data specification to the *data resolver*. The data resolver, which receives periodic *advertisements* from data sources, searches for data sources matching the data specification and returns *data-source descriptors* for those data sources. The binding manager invokes the binding module to select one of these descriptors, and passes the descriptor to the *port manager*. The port manager invokes a *port factory* to create a *port* that understands how to communicate with that kind of data source—for example, SOAP, Java Message Service (JMS) [3], JDBC, MQ, or Domino. The *security manager* ensures that the user or application creating a composer has permission to access the data source, and may provide credentials it has stored for the data source’s own authentication mechanisms. The binding manager encloses the port returned by the port manager in a *data-source handle* that is returned to the composer manager. The composer obtains data through this handle.

When selecting a data source, the binding manager also registers with the data resolver to be notified of new advertisements for that source. A data source issues such advertisements periodically, but also whenever properties of the data source (e.g., quality-of-information metrics such as freshness or precision) change. Upon receiving notification of a new advertisement for a currently bound data source, the binding manager invokes the application’s binding module, which decides either to maintain its current binding or to *rebind* to some other data source.

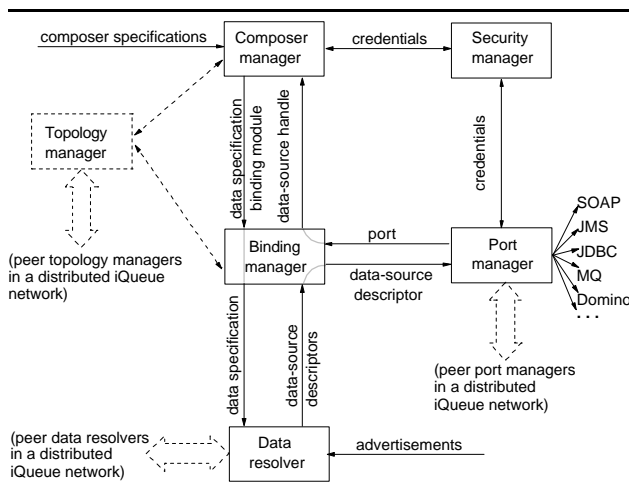


Figure 2. Internal components and data flows of iQueue. Broken lines represent components of a distributed iQueue system.

So that iQueue can scale composition to trillions of data sources, it is designed as a distributed system. We envision a federation of peer data resolvers managing a distributed data-source registry. The port manager communicates not only with external data sources but also with remote composers. Eventually, each iQueue node will include a *topology manager* to determine how computations should be subdivided into composers and where in the network each composer should execute.

3. Data model and programming model

The iQueue framework discovers and selects data sources that satisfy abstract data specifications. The data resolver matches the data specifications with advertisements made by various *member* data providers and published to the iQueue system. In the future, we expect to incorporate data from *nonmember* data providers, by providing bridges to other naming/discovery systems such as Universal Description, Discovery, and Integration (UDDI) [4], Resource Description Framework (RDF) [5], and the Service Location Protocol (SLP) [6]. To facilitate its use as a standalone discovery service, the data resolver accepts advertisements in the form of XML text, and data specifications in a simple query language derived from the SLP and Lightweight Directory Access Protocol (LDAP) [7] query languages.

3.1. Advertisements and data specifications

Advertisements have two parts:

- **Properties.** This part, similar to UDDI *yellow pages*, contains properties used to determine whether the advertised data source satisfies a given data specification. Properties potentially associated with a data source include location, data type, server load, data freshness, and data granularity. We also allow properties associated with particular kinds of data sources. The mandatory property “*identifiers*” is a set of one or more hierarchical names used for lookup.
- **Interfaces.** This part, similar to UDDI *green pages*, specifies how a client binds to the data source. The binding information includes network protocol information (e.g., SOAP or JMS), access point information (e.g., URL and port number), allowed operations, parameter descriptions, and message types. We use the Web Services Description Language (WSDL) [8] to express the binding information. WSDL currently specifies only SOAP and HTTP bindings, but we plan to extend it to express other protocol bindings, such as JMS, JDBC, and MQ.

A data source may be indexed by multiple hierarchical names, corresponding to different *roles*. For example, one application might identify a device using the name of

the registered owner, while another might identify the same device using its serial number.

A data specification for a member data source includes a set of hierarchical names, a *type scheme* indicating the names and types of properties expected to be advertised for the data sources of interest, and a boolean combination of predicates over the values of those properties.

3.2. Programming model

The central element of the iQueue programming model is a composer. Application writers create composers, and control their behavior, through a Java framework that has a class named `Composer`. Data can be both pulled and pushed from a composer. For pulling, the `Composer` method `getValue` explicitly requests the current value associated with a composer, and returns that value. For pushing, the `Composer` class has methods to subscribe and unsubscribe to *new value events*, in accordance with the JavaBeans event model [9].

The value associated with a composer is recomputed whenever its `getValue` method is called or whenever one of its data sources provides a new value. When a composer's `getValue` method is called, the iQueue system obtains the current values of all of the composer's current data sources and the composer's value is recomputed and returned. When one of a composer's data sources provides a new value, the iQueue system obtains the current values of the composer's other data sources, the composer's value is recomputed, and the composer announces the new value by invoking the new-value-event listeners that are registered with it on behalf of applications or other composers.

A `Composer` object is generated from a `ComposerSpec` object. Conceptually, a `ComposerSpec` object specifies an expression that determines how all composers generated from it compute their values. An iQueue application writer can specify the expression in a piece of declarative program text called an *eyelet*, which a *composer-specification compiler* compiles into a `ComposerSpec` object. Alternatively, a program can construct the expression tree of a composer specification out of Java objects called *fragments*. Fragments belong to classes defined in the iQueue framework, or to application-provided subclasses of abstract classes defined in the framework.

In particular, a *source-binding fragment* represents a binding to a data source. It has one input, a data-specification object. Its output is a value provided by some data source satisfying the data specification. The data specification may itself be the result of a computation. The iQueue system binds the fragment to a suitable data source, and continually rebinds the fragment as appropriate.

The framework also defines an abstract class

`JavaFragment` with the following method:

```
public abstract Object compute(Object[] inputs)
```

An application writer can insert an arbitrary computation in a composer-specification expression by defining a subclass of `JavaFragment`, overriding `compute` with a concrete method performing that computation.

We are assembling a rich library of fragment classes to help application writers build powerful composers with as little custom Java programming as possible. The library includes, for example, arithmetic operators for scalars and arrays, an if-then-else operation, a fragment that generates a new value only when the value of its input changes, and a fragment that caches the value of its input, so that it can provide its value in a top-down computation without recomputing the input.

Figure 3 shows the expression tree of a composer that generates a value when a yellow-pages data source for a user's current location contains ads for products or services on his shopping list. The `shoppingListInput` source-binding fragment provides a list-valued object from a file identified by `shoppingListDataSpec`. The `locationInput` source-binding fragment provides input from a particular location-reporting service, described by `locationDataSpec`. Each time the user's location changes, `locationInput` provides a new value to `yellowPagesDataSpecComputation`. This is an object of an application-defined subclass of `JavaFragment`, whose `compute` method takes a location as input and evaluates to a data-specification object for a yellow-pages service covering that location. The `yellowPagesInput` source-binding fragment provides a stream of yellow-pages-entry objects from the data source described by the computed data specification. The `matcher` belongs to an application-defined `JavaFragment` subclass whose `compute` method expects its inputs to be a shopping list and a yellow-pages entry. If the yellow-pages entry matches an item on the shopping list, the method returns an object representing an alert. If an application has registered a new-value-event listener with this composer,

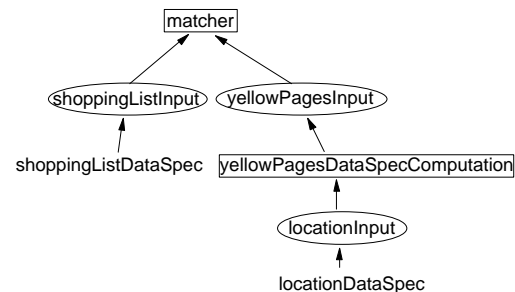


Figure 3. A composer for issuing alerts when items on a shopping list match items listed by a local yellow-pages data source. Rectangles represent Java fragments and ellipses represent source-binding fragments.

the listener is invoked with an event object that contains the alert, which the listener can then display.

4. Architecture

This section describes the architecture of the iQueue implementation. Section 2 briefly described the major components of the system—the *composer manager*, the *binding manager*, the *data resolver*, the *port manager*, and the *security manager*—and their interactions, depicted in Figure 2.

4.1. Composer manager

The composer manager is responsible for the creation of composer specifications and composers, and for the evaluation of composer expression trees. The evaluation of a source-binding fragment with a given data-specification input results in a request to the binding manager for a data-source handle compatible with the data specification. The data-source handle specifies a port for either an external data source or a composer.

Like a `Composer` object, a data-source handle has a `getValue` method and a registry of new-value–event listeners. The composer manager installs such a listener to trigger recomputation of the composer’s value when the data source generates a new value. This recomputation may entail invoking the `getValue` methods of the composer’s other data-source handles. Similarly, an invocation of the composer’s own `getValue` method can lead to the invocation of the `getValue` methods of the composer’s data source handles.

When a composer is activated, the composer manager creates an advertisement for it and transmits that advertisement to the data resolver. Depending on the access rights it specifies, the advertisement might allow the composer to be discovered as a data source and shared by other applications, or by other composers within the same application.

4.2. Binding manager

Given a data specification and a binding module specifying application binding policies, the binding manager provides the composer manager with a data-source handle bound to an appropriate data source, and rebinds the handle to new data sources as appropriate.

In determining an initial binding, and in finding a new binding for a handle, the binding manager submits a data specification to the data resolver. The data resolver returns one or more data-source descriptors for suitable data sources. The binding manager invokes the binding module to select one of these descriptors and then invokes the port manager to create a binding for the

selected descriptor.

The binding manager also registers a *data-source change listener* with the data resolver; this listener is invoked whenever the data resolver receives an advertisement changing the quality-of-information metrics of the data source. The listener invokes the corresponding binding module with an updated descriptor. If the binding module determines that the data specification should be rebound, the listener initiates the rebinding. Rebinding entails obtaining a fresh set of descriptors from the data resolver, invoking the binding module to select one of them, obtaining a new port from the port manager, and updating the data-source handle being used by the composer so that it refers to the new port. The composer continues to use the same data-source handle, unaware that the handle now refers to a new data source.

4.3. Data resolver

A data resolver receives advertisements from member data sources. Upon a request from the binding manager, it finds data sources that satisfy a given data specification and returns corresponding data-source descriptors. As explained in Section 4.2, the binding manager registers to receive data-source–change notifications from the data resolver, issued when a new advertisement for the data source reports a change in the value of some property. The data resolver must store and index information efficiently for a large number of data sources, cope with rapid service updates, and scale to a wide-area network.

Internally, the data resolver stores advertisements at the leaves of a *name tree* [10] for search efficiency. The structure of the name tree corresponds to possible values for the “*identifiers*” property (see Section 3.1). Because a data source may be indexed by multiple identifiers, an advertisement may be attached to multiple leaves of the name tree, as in the example of Figure 4.

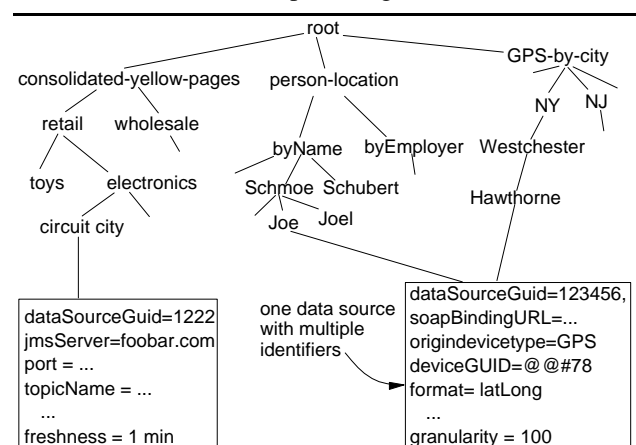


Figure 4. A name tree in a data resolver node.

In order to scale, the data resolver needs to be distributed, with various resolvers working as a federation. The distributed resolvers are similar in structure to DNS or LDAP servers. However, such technologies do not address our needs, because none of them handle rapid service updates (e.g., once a minute or more). Rapid service updates are likely to arise from market requirements for service differentiation based on dynamic quality-of-information metrics such as freshness of data. Sensor advertisements will also generate rapid service updates as sensors move, fluctuate in performance, or fail. We plan to use a high-performance, scalable wide-area content routing (publish/subscribe) network, Gryphon [11], to replicate advertisements.

4.4. Port manager

The port manager is a protocol gateway. It has a large repertoire of *port classes* for various transport mechanisms and protocols. Each port class is responsible for a particular combination of transport mechanism and usage conventions. For example, if there were a JMS-based stock-price reporting service that followed a particular convention for forming JMS topic names from stock symbols, there would be a port class for the stock service, and a different object of the class for each data source implemented as a JMS subscription to the service. The port class for the stock service, along with port classes for other JMS-based data sources, might be a subclass of an abstract class `JMSPort`, responsible for aspects common to all JMS-based data sources.

We envision that the port manager will eventually be more than a simple protocol gateway. It may cache values depending on current data requests. It may anticipate future data requests and prefetch data values. In addition, it may address data semantics, translating between data formats or filtering data on behalf of composers.

4.5. Security manager

The iQueue system must provide a number of security guarantees. It must ensure that only authorized users and applications have access to data sources. It must ensure that advertisements are seen by binding modules of only authorized computations. It must ensure that data transmitted between nodes of a distributed system is private, and that received data is authentic. It must guarantee that one application can neither learn about nor affect the execution of another application.

Our approach is to exploit existing security technology. Authorizations are granted based on access control lists. The Secure Socket Layer [12] is used when possible to protect the privacy and integrity of transmitted data.

Some external data sources require user credentials. These credentials are stored in the security manager. The

port communicating with the data source obtains them from the security manager to access the data source.

5. Implementation and initial experience

We have implemented an end-to-end prototype of the iQueue system and used it to execute several applications. The composer manager can service remote requests. It supports a wide variety of fragment classes, and the composer-specification compiler recognizes corresponding eyelet operators. The binding manager performs continual rebinding as explained Sections 2 and 4.2. The data resolver is a single name tree containing all the advertisements in the system, but able to service remote requests. The port manager supports SOAP and JMS ports, among others. The security manager is not yet implemented.

For the yellow-pages application described in Section 1.1.1, we are using two location data sources, one simulated by user clicks on a grid map and the other a location-aware Blackberry RIM pager. We have hand-crafted SOAP services for yellow-pages data sources. We are able to simulate failure of data sources. We have not yet done formal performance measurements, but initial experience shows that physical data acquisition can be a bottleneck. Location data from the pager encounters a sizable delay outside iQueue that affects the usefulness of the application. We are exploring other technologies, such as wireless-enabled GPS.

We have built an application for automated monitoring of diagnostic information from a large collection of mobile devices. We are quickly learning implications of scale and distribution that are far-reaching enough to influence our programming model. For example, in this application, it is necessary for a composer to attach to new data sources that appear *after* it has made a call to the binding manager. This implies that the data resolver must support *continuous queries*, whereby it can notify the binding manager when advertisements arrive for data sources matching a *previously submitted* data specification. This implies that iQueue composers must be able to deal with a changing number of data sources.

6. Related work

Several projects in the database community have explored the implications of querying and processing sensor data. The Telegraph project [13] is exploring adaptive query processing, in which aggregation algorithms are chosen dynamically based on properties of the input data. The historical survey in [13] traces the evolution of adaptive query processing from early attempts to select join algorithms based on daily or weekly statistics to proposals for systems that choose not

only aggregation methods, but data sources, dynamically, based on continuous feedback. Our goals for the iQueue system are similar. We take performance metrics into account when resolving data specifications and even use data from one source to construct the specification of the source from which other data will be obtained. The topology manager is aimed at adaptively replacing composer hierarchies with other, computationally equivalent, hierarchies in response to evolving system and workload conditions. The Cougar project [14] establishes distribution and data reduction as key issues in efficient querying of sensor data. The iQueue runtime system addresses data reduction and distribution while accommodating heterogeneous data access protocols and supporting abstract data specifications, data discovery, and continual binding. The NiagaraCQ project [15] addresses the problem of running continuous XQL queries on distributed XML data sources. The query splitting and grouping in NiagaraCQ may be beneficial in the design of the iQueue topology manager. However, NiagaraCQ does not support abstract data specifications or diverse data sources such as sensor data. Castro and Muntz [16] use fusion and composition of raw sensory information for interpretation of context. Our system is complimentary to systems such as these, which optimize data acquisition and make it more reliable.

Considerable attention has been paid to the use of object-oriented models for data retrieval [17]. However, our use of objects, as the basis for a composer model unifying push and pull access, is unique.

In the Intentional Naming System [10], resources to be discovered are described by abstract specifications. Although our notion of data specifications and the design of our data resolver (particularly the use of name trees) are largely inspired by this system, we support wide-area discovery using the Gryphon publish-subscribe system [11], a robust, scalable implementation of the Java Message Service API. The notion of specifying data sources in terms of properties of the data rather than properties of the source can also be found in the work of Estrin *et al.* [18] on *data-centric* sensor networks.

Like iQueue, the Amit system [19] is a system for building reactive applications. Amit processes streams of *events*, recognizes patterns indicative of specified *situations*, and responds with appropriate actions. We envision complementary roles for Amit and iQueue. The iQueue system can gather data from a wide variety of data sources and direct it to composers. A composer can be written that directs incoming data to Amit, with each arrival of a new value treated as an Amit event. Amit can examine the event stream, recognize situations, and respond to them.

7. Conclusions and ongoing work

A data composition framework like iQueue can be of great help writing applications that use dynamic, pervasive data. The application writer's task is made easier by automated binding and rebinding of abstract data specifications; system support for the mechanics of using a variety of data sources, with different protocols and formats; and a programming model based on hierarchies of composers whose behavior is specified by expressions for computing values that can be either pulled or pushed. Initial experience with our prototype is encouraging.

Our long-range goal for iQueue is to build a system that will address most of the pervasive-data-composition challenges we identified in [1]. We are working on a number of these challenges:

- **A scalable, distributed iQueue system:** The iQueue system must eventually scale to compose data from trillions of sensors. Such scaling requires a finely tuned distributed system. Distribution provides fault tolerance, enables load balancing, and facilitates placement of data-reduction computation to minimize network traffic. We have built remote data resolvers and composer managers. We are beginning design of a topology manager that controls a logical composition topology over a physical network, combining compiler optimization and database query-optimization techniques to tune the composition topology dynamically.
- **Data-source description and discovery.** Currently, iQueue supports data specifications and data resolution only for *member* data sources (that actively advertise to iQueue). Eventually, we expect to support a variety of important web resource description schemes, and to discover network-accessible *nonmember* data sources. We envision translating certain data specifications from one naming scheme to another in search of suitable data sources. We also expect to *synthesize* some data sources as part of the resolution process, finding data sources that *nearly* satisfy data specifications, for which iQueue can find or construct composers to adapt the data to the required form.
- **Security.** The iQueue system has security challenges beyond authentication and encryption. For example, some composers may be trusted to read detailed data to which access is highly restricted, and to generate “sanitized” summary data that can be made more widely accessible. Also, because iQueue will host composers for multiple applications, we must guarantee privacy and fair resource sharing among composers.
- **Programming abstractions.** As we write new iQueue applications, we are discovering many recurring patterns. We are growing our library of fragment classes, in the hope that application-provided `JavaFragment` subclasses will rarely be needed. In conjunction with this effort, we are actively refining

and enhancing our declarative programming model for composition, eyelets. We believe that eyelets will dramatically simplify the development of composition-based applications. A detailed exposition of the eyelet model is currently in preparation.

References

- [1] Norman H. Cohen, Apratim Purakayastha, John Turek, Luke Wong, and Danny Yeh. Challenges in flexible aggregation of pervasive data. IBM Research Report RC 21942, January 23, 2001, <http://www.research.ibm.com/sync-msg/RC21942.pdf>
- [2] Simple Object Access Protocol (SOAP) 1.1 W3C Note 8 May 2000. <http://www.w3.org/TR/SOAP>
- [3] Sun Microsystems. Java Message Service API. <http://java.sun.com/products/jms>
- [4] UDDI Technical White Paper, uddi.org, September 6, 2000, http://uddi.org/pubs/Iru_UDDI_Technical_White_Paper.pdf
- [5] Resource Description Framework (RDF) Model and Syntax Specification. W3C Recommendation REC-rdf-syntax-19990222, February 22, 1999, <http://www.w3.org/TR/REC-rdf-syntax/>
- [6] E. Guttman, C. Perkins, J. Veizades, and M. Day. Service Location Protocol, Version 2. RFC 2608, June 1999, <ftp://ftp.isi.edu/in-notes/rfc2608.txt>
- [7] T. Howes, The String Representation of LDAP Search Filters, RFC 2254, December 1997, <ftp://ftp.isi.edu/in-notes/rfc2254.txt>
- [8] Web Services Description Language (WSDL) 1.1 W3C Note 15 March 2001. <http://www.w3.org/TR/wsdl/>
- [9] Graham Hamilton, ed. *JavaBeans*, version 1.01. Sun Microsystems, July 24, 1997, <http://java.sun.com/products/javabeans/docs/beans.101.pdf>
- [10] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99), December 12-15, 1999, Kiawah Island Resort, South Carolina, published as *Operating Systems Review* **33**, No. 5 (December 1999), 186-201
- [11] Guruduth Banavar, Tushar Chandra, Bodhi Mukherjee, Jay Nagarajarao, Robert E. Strom, Daniel C. Sturman. An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems. Proceedings of the International Conference on Distributed Computing Systems, 1999
- [12] Netscape. SSL 3.0 Specification. <http://home.netscape.com/eng/ssl3/>
- [13] Joseph M. Hellerstein, Michael J. Franklin, Sirish Chandrasekaran, Amol Deshpande, Kris Hildrum, Sam Madden, Vijayshankar Raman, and Mehul A. Shah. Adaptive query processing: technology in evolution. *IEEE Data Engineering Bulletin* **23**, No. 2 (June 2000), 7-18, <http://www.research.microsoft.com/research/db/debull/A00june/hellerstein.ps>
- [14] Ph. Bonnet, J. Gehrke, P. Seshadri. Towards Sensor Database Systems. Second International Conference on Mobile Data Management. Hong Kong. January 2001
- [15] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: a scalable continuous query system for Internet databases. Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 15-18, 2000, Dallas, Texas, 379-390
- [16] Paul Castro and Richard Muntz. Managing context data for smart spaces. *IEEE Personal Communications* **7**, No. 5 (October 2000), 44-46
- [17] Omran A. Bukhres and Ahmed K. Elmagarmid, eds. *Object-Oriented Multidatabase Systems: A Solution for Advanced Applications*. Prentice Hall, Englewood Cliffs, New Jersey, 1996
- [18] Deborah Estrin, Ramesh Govindan, John Heidemann, and Satish Kumar. Next century challenges: scalable coordination in sensor networks. Proceedings of the fifth annual ACM/IEEE international conference on mobile computing and networking, Seattle, Washington, August 1999, 263-270
- [19] Asaf Adi, David Botzer, Opher Etzion, and Tali Yatzkar-Haham. Push technology personalization through event correlation. In Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang, eds., *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, Morgan Kaufmann, San Francisco, 2000, 643-645, <http://www.vldb.org/conf/2000/P643.pdf>