RC23388 (W0410-156) October 22, 2004
Computer Science

IBM Research Report

Building Context-Aware Applications with Context Weaver

Norman H. Cohen, James Black, Paul Castro, Maria Ebling, Barry Leiba, Archan Misra, Wolfgang Segmuller
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Building Context-Aware Applications with Context Weaver

Norman H. Cohen, James Black*, Paul Castro, Maria Ebling, Barry Leiba, Archan Misra, and Wolfgang Segmuller

IBM Thomas J. Watson Research Center
Hawthorne, New York

Abstract. Context Weaver is a platform that simplifies writing of context-aware applications. All providers of context information registered with Context Weaver provide data to applications through a simple, uniform interface. Applications access data sources not by naming particular providers of the data, but by describing the kind of data they need. Context Weaver responds with providers of context information that may include not only devices, services, and databases external to Context Weaver, but also programmed entities that process context information from other providers. If a provider fails, Context Weaver automatically tries to rebind the application to another provider of the same kind of data. Privacy policies, specified partly by administrators and partly by individuals who are the subject of context information, are enforced by Context Weaver to protect the privacy of those individuals. A health-care application developed with Context Weaver helps nurses monitor the conditions of their patients.

Keywords: C.3.h Ubiquitous computing, D.2.2.c Distributed/Internet based software engineering tools and techniques, D.2.6. Programming Environments/Construction Tools, D.2.17.i Programming paradigms

Context-aware applications

Our day-to-day living experiences, the activities of an individual worker, and the processes of a large corporation can all be simplified by computing systems that are aware of an individual's *context*. By context, we mean information about an individual and his surrounding environment that may be used to deduce the ways in which the computing system can best serve the individual. This deduction can be made without active input from the individual, thus preserving the most precious of all resources—the individual's attention—for other tasks.

Context may include an individual's location, calendar appointments, blood pressure, or current activities. An individual's context may also include traffic conditions, airline schedules, the weather, or the set of people in the same room. Low-level fragments of context information—for example, that five people are located within a few feet of coordinates (1,10,20) in a building coordinate system, that those coordinates correspond to Conference Room A, and that Conference Room A is currently scheduled for a budget meeting—may be composed to deduce higher-level information—for example, that each of those individuals is currently involved in a budget meeting.

* On sabbatical from the University of Waterloo, Waterloo, Ontario, at the time this paper was written.

Examples of context-aware applications include advising a driver to take a particular route based on his location, his destination, and current traffic conditions; advising a nurse to attend to a particular patient based on the medical telemetry being received from all patients on a ward; and delivering a message either by cell phone or by e-mail depending on the recipient's current context. The individuals who benefit from context-aware applications may not be sitting with a keyboard, mouse, and display, and may in fact be engrossed in other activities. They may remain unaware of the computer systems working on their behalf except when those systems interrupt them for some urgent purpose.

The challenges of writing context-aware applications

Context-aware applications are difficult to write, for several reasons. First, sources of context information vary widely. Second, some sources of context information are unreliable, and may have to be replaced dynamically with other sources. Third, the task of composing low-level context information to deduce high-level context information can be complex.

Sources of context information include, among others, sensors, web services, publish-subscribe systems, instant-messaging systems, and relatively static repositories such as databases and calendars. Different sources of context information provide data in different formats, using different units of measurement, according to different protocols. Some sources actively push data to subscribers while others passively provide data when it is pulled.

The power of context-aware applications stems in part from the availability of large numbers of inexpensive devices. Some devices—for example, single-use devices deployed at a disaster site—are intentionally designed to be cheap rather than robust. Other devices may be individually reliable, but deployed in such large numbers that *some* device is likely to fail over a given period of time. In either of these cases, redundant sources of context information are generally available. Sometimes the role of a faulty sensor can be taken over by an identical nearby sensor. Sometimes an alternative source of context information can be employed, as when a road-loop traffic sensor fails and a traffic estimate from analysis of video images is used instead. A context-aware application based on fragile sources of context information must be prepared to replace one source of context dynamically with another, perhaps based on a different kind of raw information.

The composition of context information is complex in part because some of the information comes from active ("push") data sources and some from passive ("pull") data sources. Polling passive data sources at regular intervals can result in excessive processor and network load if the polling interval is too short, or in stale data if the polling interval is too long. Some active data sources may generate data at inopportune times, when the data is not really needed. The data must then be cached for later use or discarded. The orchestration of asynchronous events—the arrival of data from active data sources and application demands for freshly composed data—is itself a difficult and error-prone programming problem.

Simplifying application development with Context Weaver

Context Weaver simplifies the writing of context-aware applications by addressing each of the difficulties we have discussed. All sources of context information registered with a Context Weaver installation provide data to applications through a simple, uniform interface. Applications access data sources not by naming particular providers of the data, but by describing the kind of data they need, and Context Weaver searches for an available source of such data. If the source fails, Context Weaver automatically rebinds the application to another provider of the same kind of data, if there is one. Providers of context information include not only devices, services, and databases external to Context Weaver, but also programmed entities called composers, which compose context information from other data providers. Context Weaver applications use composers and external sources of context information interchangeably. The behavior of a composer is specified using building blocks that deal in well-defined ways with both passive and active providers of input context information. This approach frees the application developer from timing concerns, freeing her to concentrate on the logical mapping from lower-level context information to higher-level composed information.

Writing a Context Weaver application

A Context Weaver application includes client code, typically written in the Java language, that issues a query for data providers, obtains data providers in response to the query, and processes the data they provide. The Java code may be invoked from a standalone application, an applet, a servlet, an Enterprise Java Bean, or Java Server Pages, for example. The query submitted by the application requests one provider or a list of providers of a specified kind, activated with specified values, currently satisfying specified conditions. A Context Weaver application also includes composers. Some composers may be programmed for a specific application, while others may be part of a library of composers useful in a specific application domain, or a library of general-purpose composers.

Obtaining and using data providers

A Context Weaver application can issue a *provider query* to Context Weaver requesting access to a source of some particular kind of context information. Context Weaver responds with a list of zero or more handles to *data providers* that satisfy the query. Each handle may be accompanied by a descriptor enumerating properties of the data provider.

A data provider is regarded as always having a current value. In addition, some data providers will, from time to time, generate new values. Through a data-provider handle, an application may request a data provider's current value. In addition, an application can register a listener with a data provider that will be invoked each time the data provider generates a new value.

Even though some sources of context information are passive and some are active, this simple interface accommodates any source. When an active source is asked for its current value, it will return the value that it generated most recently. A listener for generated values can be registered with a passive data source, but it will never be called.

Values are represented as Java objects implementing an interface named Data. Data objects can be mapped to and from XML documents, but the Data interface provides a higher level of abstraction than the XML Document Object Model (DOM) [1]. In particular, a Data object can represent a list of values. Each value represented by a Data object belongs to some type. The type system is based on XML Schema [2]. We base our data model on XML because we anticipate that many emerging sources of context information will provide data in that form.

The form of a provider query

A Context Weaver provider query is a descriptive name [3]. That is, it describes what kind of context information is sought, not where the information is to be obtained. The use of a descriptive name allows Context Weaver to select the best available source of context information satisfying the query. The selection may be based, for example, on the current values of fluctuating quality-of-service or quality-of-information metrics. An application based on descriptive names rather than requests to a particular source of context information can tolerate the failure of a source, as long as there are other sources available to which the same descriptive name applies. New sources of context information can be registered with Context Weaver, and obsolete sources can be removed, without modifying an application that uses descriptive names. An application written for one Context Weaver installation can be ported directly to another installation, in which different sources of the required context information have been registered.

A source of context information has a descriptor enumerating its properties. A provider query is, in part, a test of these properties. In addition to static properties, such as the resolution of a camera, the descriptor may include dynamic properties, such as the current ticker-tape delay of a stock-price. One dynamic property present in every descriptor is a snapshot of the current value of the data provider it describes. Thus a provider query can ask, say, for all providers of room-temperature readings that are currently reporting temperatures outside the normal range.

Every source of context information registered with Context Weaver is registered as belonging to some particular *provider kind*. For example, there could be a provider kind for sources of a person's location in terms of a building coordinate system, given the person's badge number, or for sources of a person's instant-messaging status, given the person's user ID. Different context-information sources of the same provider kind may have radically different implementations. For example, one source of a person's location within a building might be based on RFID badge readers, and another might be based on entries in the person's appointment calendar.

A provider query has four components. The first is the name of a provider kind. The second is a predicate to be applied to the properties in a descriptor. The third is a set of values for *activation parameters*. The fourth is a *selection mechanism* determining which data providers, among those that match the query, should be returned.

The provider kind determines the type of data that is sought, the descriptor properties that may be queried in the predicate, and the types of values that may be used for activation parameters. Implicitly, each provider kind is associated with underlying semantics, including the relationship of a data provider to its activation parameters. We

do not formalize this semantic relationship; rather, we assume that the writer of a provider query is familiar with the semantics of various provider kinds, just as the writer of a method call in a Java program is familiar with the semantics of various methods.

The predicate in a provider query is a boolean-valued XQuery [4] expression that is applied to the XML representation of a provider descriptor. The XQuery expression language is rich, allowing arbitrary boolean expressions that combine arbitrary tests on property values.

Activation parameters provide the data that may be needed to initialize, or to establish a connection with, a context-information source of a given provider kind. Each such source may use the same activation parameter in a different way. For example, one source of stock prices, given a stock-symbol activation parameter, might use the activation parameter as a key in a database lookup; another might use it to construct a topic name for a Java Message Service subscription.

Currently, the selection mechanism has one of two values—either *all* or *any one*—indicating either that all matching data providers should be returned or that one matching data provider, arbitrarily chosen by Context Weaver, should be returned. We envision a more flexible selection mechanism for future versions of Context Weaver, consisting of two parts. The first part is an integer-valued XQuery expression that can be applied to the XML representation of a provider descriptor to obtain a *score*. The second part is a policy determining how scores are to be used to select the sources of context information that are to be returned. One possible policy is to return the *n* highest-scoring sources for some value of *n*; another possible policy is to return all sources with scores greater than some specified threshold.

Composers

It is unrealistic to expect that all the context information required by a context-aware application can be provided by the external sources registered with Context Weaver. Therefore, Context Weaver also has *composers*—data providers that work by obtaining inputs from other data providers and performing computations to determine their own current values. A given composer may act as an active data provider, a passive data provider, or both. A simple composer might translate the context information from another data provider into a different format or unit of measurement. A more sophisticated composer might filter the stream of values generated by an active source, or act as an active source by periodically polling a passive source and generating the resulting value. Some composers obtain low-level information from two or more other data providers and apply a formula to obtain a new, higher-level kind of information.

A data provider that Context Weaver returns to an application in response to a provider query may be either an external source or a composer. The distinction is invisible to the application. Either kind of data provider may be queried for its current value and may have a listener registered with it to react to the generation of new values.

Likewise, the data providers from which a composer obtains its inputs may include any combination of external sources and other composers. A data provider returned to an application in response to a provider query, if it is not an external data source, is the root of a tree of data providers like that shown in Figure 1. Every composer

is an instance of a *composer specification*, a template that determines the behavior of the composer. A Context Weaver installation has a repository of composer specifications that may be used to satisfy provider queries. This repository may include both general-purpose composer specifications useful in particular application domains and special-purpose composer specifications written by application developers to simplify the writing of an application.

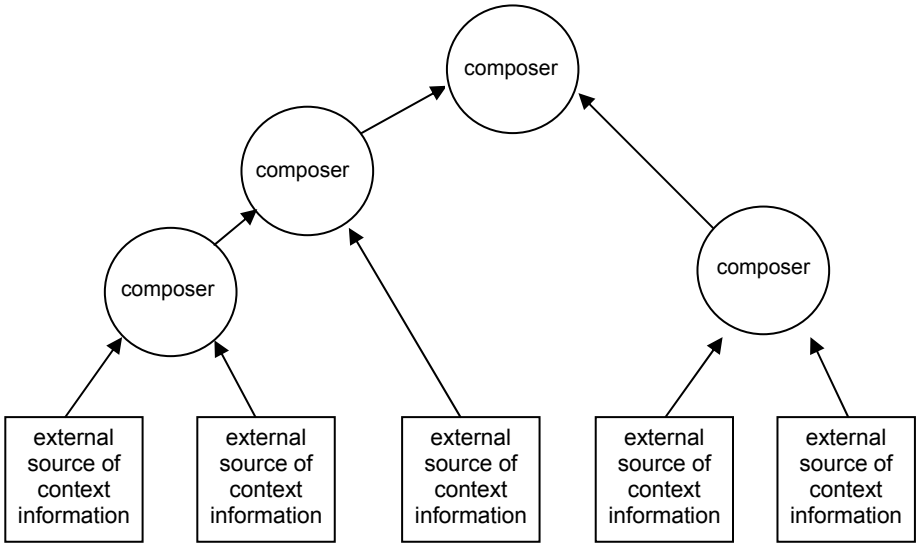

Figure 1. A tree of data providers. The leaves of the tree are external data sources and all other nodes of the tree are composers.

A composer specification is simply an expression. The value of the expression determines the current value of the composer. The operators found in these expressions include *input* operators. Input operators contain provider queries, and represent values obtained from data providers satisfying the queries. Figure 2 depicts a simple composer-specification expression, for a composer that converts Celsius temperatures obtained from another data provider into Fahrenheit temperatures. (Most composers are more intricate.) This composer acts as a passive data provider when its current value is requested; the expression is evaluated to determine its current value, and the evaluation of the input operator causes a current value to be requested from some other data provider. The composer acts as an active data provider when the input operator's data provider emits a new value; the expression is evaluated using that value and the composer emits the result of that evaluation. Composer-specification expressions can be written the iQL language described in [5] and compiled into the form stored by Context Weaver.

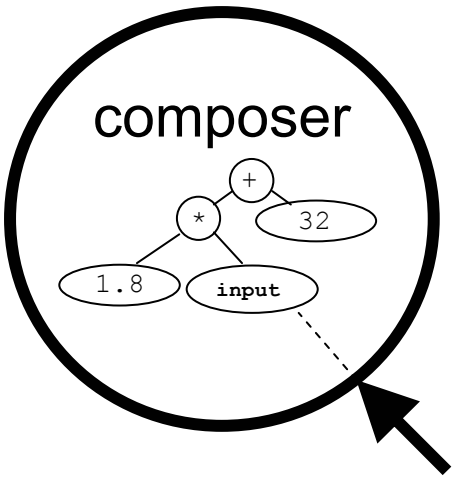

Figure 2. A composer-specification expression. The outer circle corresponds to one node in a tree of data providers like that in Figure 1, and the arrow represents an input from another data provider. The tree inside the circle represents a composer-specification expression; the nodes of the tree are operators. The input operator corresponds to values obtained from the data provider (not shown) at the tail of the arrow.

In addition to the input operator and the usual arithmetic and logical operators, composers may include operators unique to the task of data composition. A *polling operator* evaluates an operand periodically and emits the result, allowing active data providers to be built out of passive ones. A *filtering operator* emits the value of its operand only if specified conditions are satisfied. A *caching operator* stores its result, avoiding redundant requests for the current value of another data provider. A *merging operator* uses the value of any of several operands as its own value. A *previous-value operator* yields the value that its operand had on the previous evaluation of the composer, allowing composers to maintain state. *Compound-event operators* emit values when they recognize sequences of values arriving in particular patterns. *Native operators* invoke specified Java methods. Further details on these and other operators can be found in [5].

The provider query in an input operator may be constructed dynamically, based on the values of other subexpressions of a composer-specification expression. When the provider query of an input operator changes, it is reprocessed. In addition, a polling operator can cause an unchanged query to be reprocessed periodically. The reprocessing of a query may cause the input operator to be bound to different data providers over time. This can happen, for example, if a previously found source of context information is no longer available, or if the query tests a quality-of-service attribute in the provider descriptor and the quality of service has degraded since the query was last processed.

Context Weaver manages the evaluation of expressions at the appropriate time, and the delivery of results to the proper places. The application developer need not be concerned with coordinating asynchronous events, and can concentrate on the logical mapping of input values to output values. By dynamically reprocessing provider queries in input operators, Context Weaver frees the application developer from the task of monitoring sources of context information to ensure that they are still providing an acceptable quality of service.

Enabling sources of context information

Before an external source of context information can be returned in response to a provider query, it must be registered with Context Weaver. Before it can be registered, it must have a driver that implements an interface understood by Context Weaver. The driver consists of two parts, an *adapter* and an *activator*. The adapter must implement a method to report the current value of a context-information source. In the case of an active source, the adapter is also responsible for calling a particular method whenever the source generates a new value. The job of an activator is to examine activation parameters and to construct and initialize an adapter object. In the process, the activator may perform such setup actions as initializing a device, connecting to a web service, or subscribing to a JMS topic.

The success of Context Weaver will depend in large measure on the availability of many context-information sources. To that end, we have embarked on the implementation of a wide variety of drivers. Context Weaver drivers for the Blackberry PDA (providing information about the location and activities of its user), an instant-messaging service, a PC desktop (indicating which window currently has the focus), active-badge-based location information, network printer status, appointment calendars, telephone status, wireless LAN (indicating the access point with which an individual's wireless LAN device is currently associated), and weather reports are implemented or under development.

Ensuring privacy

Context information about an individual (the *subject* of the information) is potentially very sensitive. Any system like Context Weaver must protect against unauthorized access to the information. We believe that the access controls for context information belong to the subject of the information, in general, and have developed a Context Privacy Engine (CPE) to support privacy policies.

CPE privacy policies allow specification of access controls for individual requestors, for user groups, and for system groups. In addition to traditional access control (e.g., "User A can access information X"), the policies incorporate P3P [6], and may themselves be based on the context of the requestor or of the subject (e.g., "Members of the Management group can access my location if I am not on vacation and the requestor is in the office"). Control is generally in the hands of the subject, but there are system default policies, allowing administrative control of a base setup that users can work from, and system override policies, allowing administrative mandates when necessary (e.g., "Members of the Dispatcher group can access the locations of members of the DeliveryDrivers group if the subject is on the job").

A Context Weaver privacy policy depends, in part, on the *requesting function*, which may be a client application or a composer. A requesting function that uses information to make other determinations or perform useful services, without actually exposing the information it has used, might be granted access to sensitive information. In contrast, a function that potentially exposes that information might be denied access.

For example, a composer that gives a subject's "current telephone number" uses the subject's location as input to select an appropriate telephone number depending upon whether the subject is at the office, at home, visiting her parents, or on the road. The subject does not want people in her Customers group to know her location, but is willing to allow them to use the composer to find the best number to telephone when they want to talk with her. Context Weaver can implement privacy controls like that, through CPE.

Early experience

We have been developing Context Weaver applications to test the usefulness of Context Weaver in practice. One of these applications is a hospital scenario in which a nurse uses a web portal to monitor the condition of patients in the ward and attend to them in the most appropriate manner. The portal includes a triage portlet listing patients sorted by the urgency of their alerts. There is a second portlet that provides additional detail for a single patient on request.

In the implementation of this application, depicted in Figure 3, each patient monitor is a Context Weaver external data source for a (simulated) medical device measuring quantities like blood pressure or body temperature. For each patient, there is a composer that collects raw context information from that patient's vital-sign monitors and aggregates it into a vector. A second composer for each patient analyzes this vector and emits alerts when the combination of vital signs is a cause for concern. Each patient portlet receives data from both the aggregation composer and the alert composer corresponding to that patient. Another composer collects the data emitted by the various patient alert composers and emits a list of patients. The triage portlet receives data from this composer and sorts it by alert severity.

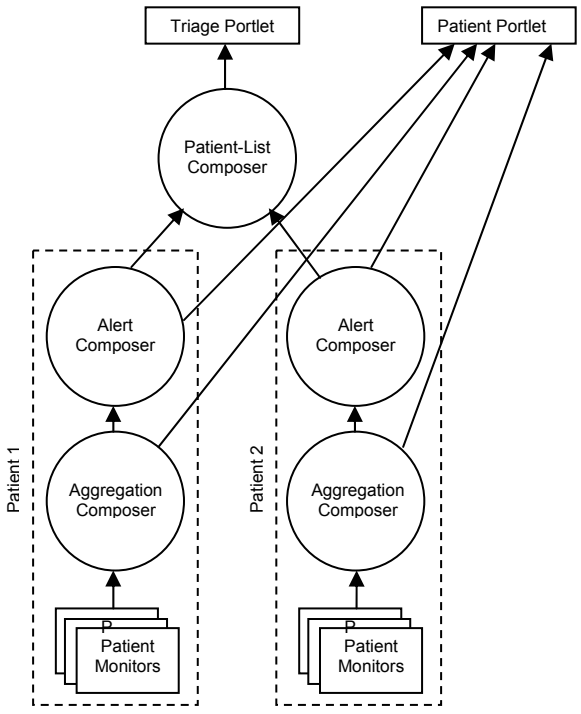

Figure 3. Implementation of the nurse portal application.

Our experience developing this application highlighted the need for additional debugging facilities and tools. A common symptom of an application error was a provider query that failed to find any matching data providers. In some cases the cause was an incorrectly installed activator class, provider kind, or provider. In other cases the cause was an exception thrown by an activator. Better diagnostic output, tracing the steps in the discovery and activation of a source of context data, would make it easier to find errors of the first kind, but a mature set of administrative tools, automating the installation of activators, providers, provider kinds, and composers would avoid such errors in the first place. Errors of the second kind could be more effectively rooted out by a testing environment allowing context-source drivers and composers to be tested in isolation, using simulated input data for the composers. Besides disentangling the sources of errors, such an environment would facilitate parallel development of drivers and composers.

We also learned that the interface between data-provider handles and portlets is not as smooth as it ought to be. In particular, initialization of portlets was awkward. We anticipate that boilerplate glue code will reduce the friction between the Context Weaver programming model and the programming models of Context Weaver clients.

Related work

A Context Weaver composer is a special case of what Wiederhold [7] calls a *mediator*. Mediators consume raw, low-level data and produce refined, higher-level data, and may be arrayed in a hierarchy. A similar hierarchy can be found in the Solar [8] system. In both cases, the hierarchy is constructed statically, and an application explicitly invokes a specific node in the hierarchy. In contrast, a Context Weaver application issues a descriptive provider query, and Context Weaver dynamically constructs an appropriate hierarchy of data providers.

Like our provider queries, the data-centric names of [9] and the intentional names of [10] are descriptive names used to discover devices or services available on a network. However, intentional names are compared to actual data sources on the basis of exact matches of attribute values. The Ninja project's Service Discovery Service [11] and the Jini [12] Lookup Service are also restricted to conjunctions of attribute equality tests. The XQuery predicates used by Context Weaver are more powerful than conjunctions of attribute equality tests. For example, given a provider descriptor whose properties include the *x* and *y* coordinates of some source of context data, it is straightforward to write an XQuery predicate that returns true if and only if the source is located in a specified rectangle. An enhanced version of data-centric names [13] replaces attribute equality tests with comparisons each consisting of an attribute name, a comparison operator, and a constant value, but allows such comparisons to be combined only by conjunction. With this approach, it remains impossible to test whether a data source with given *x* and *y* coordinates is located in a complex polygon comprising multiple rectangles, for example.

The Intentional Naming System [10] has metrics for choosing among discovered services, but these metrics consist of a single number for each kind of resource. There is no way for one application to query for the nearest printer and another application to query for the fastest one. The enhanced selection mechanism we envision for future

versions of Context Weaver is more versatile, allowing each application to compute its own preference metric from arbitrary combinations of data-source properties.

Conclusions

Our early experience has pointed out rough edges that need to be smoothed out to make the Context Weaver application-development experience as pleasant as it has the potential to be. We are currently focusing on tools to address the problems that emerged in test applications. We are also developing new context-source drivers and applications. We expect these efforts to yield new insights that will help us refine Context Weaver further.

As more sources of context information become widely deployed, it will be increasingly important to be able to develop context-aware applications quickly, cheaply, and reliably. By accepting a descriptive provider query that might be satisfied in one environment by a sensor, in another environment by a web service, and in yet another environment by a programmed computation, Context Weaver makes it easier to write an application that will port to a wide variety of environments. Composer specifications for general-purpose or domain-specific computations can be reused to simplify numerous applications. Similarly, Context Weaver adapters and activators allow the effort needed to interface with a particular source of context information to be amortized over a large number of applications.

Context Weaver also simplifies the writing of context-aware applications by taking care of administrative details. These details include finding appropriate data providers, rebinding to new data providers when previously bound data providers fail or otherwise become inappropriate, and orchestrating asynchronous events in compliance with a nonprocedural specification. With these details taken care of, the application developer can concentrate on application-specific logic.

Acknowledgements

Marion Blount, John S. Davis II, William Jerome, Xuan Liu, and Apratim Purakayastha made invaluable contributions to the discussions that led to Context Weaver and CPE.

References

[1] A. Le Hors, P. Le Hégaret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrne, “Document Object Model (DOM) Level 2 Core Specification, Version 1.0,” W3C Recommendation, Nov. 13, 2000, http://www.w3.org/TR/DOM-Level-2-Core/.

[2] P.V. Biron and A. Malhotra, eds., “XML Schema Part 2: Datatypes,” W3C Recommendation, May 2, 2001, http://www.w3.org/TR/xmlschema-2/.

[3] M. Bowman, S.K. Debray, and L.L. Peterson, “Reasoning About Naming Systems,” *ACM Trans. Programming Languages and Systems*, vol. 15, no. 5, Nov. 1993, 795–825.

[4] S. Boag, D. Chamberlin, M.F. Fernandez, D. Florescu, J. Robie, and J. Siméon, “XQuery 1.0: An XML Query Language,” W3C Working Draft, May 2, 2003, http://www.w3.org/TR/xquery/.

[5] N.H. Cohen, H. Lei, P. Castro, J.S. Davis II, and A. Purakayastha, "Composing Pervasive Data Using iQL," *Proc. 4th IEEE Workshop Mobile Computing Systems and Applications* (WMCSA 2002), IEEE CS Press, pp. 94–104.

[6] L. Cranor, M. Langheinrich, M. Marchiori, M. Presler-Marshall and J. Reagle, "The Platform for Privacy Preferences 1.0 (P3P1.0) Specification," W3C Recommendation, Apr. 16, 2002, http://www.w3.org/TR/P3P/.

[7] G. Wiederhold, "Mediators in the Architecture of Future Information Systems," *IEEE Computer*, vol. 25, no. 3, Mar. 1992, pp. 38–49.

[8] G. Chen and D. Kotz, "Context Aggregation and Dissemination in Ubiquitous Computing Systems," *Proc. 4th IEEE Workshop Mobile Computing Systems and Applications* (WMCSA 2002), IEEE CS Press, pp. 105–114.

[9] C. Intanagonwiwat, R. Govindan, and D. Estrin, "Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks," *Proc.6th Annual Int'l Conf. Mobile Computing and Networking* (MobiCom 2000), ACM Press, pp. 56–67.

[10] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley, "The Design and Implementation of an Intentional Naming System," *Proc. 17th ACM Symp. Operating Systems Principles* (SOSP '99), *Operating Systems Review* vol. 33, no. 5, Dec. 1999, pp. 186–201.

[11] S.E. Czerwinski, B.Y. Zhao, T.D. Hodes, A.D. Joseph, and R.H. Katz, "An Architecture for a Secure Service Discovery Service," *Proc. 5th Annual ACM/IEEE Int'l Conf. Mobile Computing and Networking* (MobiCom '99), ACM Press, pp. 24–35.

[12] Sun Microsystems, "Jini Technology Core Platform Specification, Version 2.0," June 2003, http://wwws.sun.com/software/jini/specs/.

[13] J. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, and D. Ganesan, "Building Efficient Wireless Sensor Networks with Low-Level Naming," *Proc. 18th ACM Symp. Operating Systems Principles* (SOSP 2001), ACM Press, pp. 146–159.