Descriptive Naming of Context Data Providers

Norman H. Cohen, Paul Castro, Archan Misra

IBM T. J. Watson Research Center
19 Skyline Drive, Hawthorne, New York 10532, USA
{ncohen, castrop, archan}@us.ibm.com

Abstract. Much context data comes from mobile, transient, and unreliable sources. Such resources are best specified by descriptive names identifying what data is needed rather than which source is to provide it. The design of descriptive names has important consequences, but until now little attention has been focused on this problem. We propose a descriptive naming system for providers of context data that provides more flexibility and power than previous naming systems by classifying data providers into "provider kinds" that are organized in an evolving hierarchy of subkinds and superkinds. New provider kinds can be inserted in the hierarchy not only as subkinds, but also as superkinds, of existing provider kinds. Our names can specify arbitrary boolean combinations of arbitrary tests on data-source attributes, yielding expressive power not found in naming schemes based on attribute matching.

1. Introduction

A number of systems obtain services from network resources such as sensors, cameras, printers, cellular phones, and web services. These resources may be mobile, they may be ephemeral, and their quality of service may fluctuate. It has become widely accepted that such systems should not require users to name a specific resource from which they wish to obtain services, but rather, to describe what the resource is expected to provide. This approach, known as descriptive [1], data-centric [2], or intentional [3] naming, allows a name resolver to discover an appropriate resource at runtime.

Descriptively named resources are important in context-aware computing, because a context-aware application often requires a particular type of context information rather than information from a particular source. For example, the application may require the location of a particular individual; it should be possible for the application to ask for this location without considering whether it is deduced from the location of the individual's cell phone, laptop computer, or car. Descriptive naming allows a name resolver to select the best available provider of data, based on current conditions, and to select a new provider when those conditions change. It makes an application robust against the failure of any one device or service. It facilitates the frequent addition or removal of context-data providers without modification of applications that use them. It allows an application to be

ported easily to an environment with a different set of context-data providers.

To accommodate a wide variety of applications, the scheme for writing a data-provider query must be flexible enough to describe any provider of context data. Different applications may need to query, for example, for providers of Fahrenheit temperatures at a given latitude and longitude, Celsius temperatures of the patient in a given hospital bed, current prices of IBM stock in U.S. dollars, the number of the room where a given active badge was last sensed, and the identification numbers of all vehicles in a specified zone with excessive engine temperatures. Clearly, it is untenable to establish a fixed vocabulary of concepts and data types to be used in provider queries.

The rest of this paper is organized as follows. Sect. 2 summarizes our approach, explaining what is unique about it and why it is preferable to previous approaches. Sect. 3 describes the programming model that we assume for context applications. Sect. 4 explains the nature of our descriptive names and the underlying model. Sect. 5 describes a prototype implementation of our naming system. Sect. 6 discusses related work and Sect. 7 presents our conclusions. The focus of the paper is the proposed naming model, not the prototype implementation. We believe that there must be fundamental improvements to the structure and semantics of names for providers of context data before a scalable and continually upgradeable context infrastructure can be practically deployed.

2. Our Contribution

The design of descriptive names for providers of context data has profound consequences for the expressiveness of queries, the efficiency of name resolution, and the ability of the naming system to grow to accommodate previously unforeseen kinds of resources. Nonetheless, until now, little attention has been focused on this issue, or on the consequences of various design decisions. We have explored the issues to be considered in designing a descriptive naming system for sources of context data, developed a descriptive naming scheme, and implemented a prototype system that resolves our descriptive names.

To name a data provider, we must be able to specify the kind of data we need—for example, Celsius temperature readings or the price of IBM stock—and the constraints we place on providers of that kind of data—for example, that the temperature readings be taken within a specified region, or that the stock price reflect a ticker delay of no more than twenty minutes. To identify the kinds of data we need, we propose a vocabulary of *provider-kind names*. To express constraints, we propose a combination of parameters, predicates on attributes of individual data providers, and filters for selecting from among eligible data providers according to application-specified metrics.

One challenge in obtaining a useful vocabulary of provider-kind names is to allow queries with varying degrees of specificity. For example, it should be possible for one application to request data providers giving at least the latitude and longitude of a vehicle with a given identifier, and for another to request data providers giving at least the latitude, longitude, and elevation of such a vehicle. Any query for providers of latitude and longi-

tude should be satisfied by providers of latitude, longitude, and elevation. Another challenge is to enable the vocabulary to evolve in a disciplined way—to incorporate new kinds of data providers unrelated to previously existing kinds, to add a specialization of an existing provider kind, or to add a generalization of two or more existing provider kinds (so that a query for providers of the new kind can be satisfied by providers of any of the existing provider kinds). We address these challenges by organizing provider kinds in a multiple-inheritance hierarchy of subkinds and superkinds, analogous to the hierarchy of subclasses and superclasses in an object-oriented programming language. Our hierarchy is novel in that a new provider kind can be stipulated to be both a subkind of certain existing provider kinds and a superkind of certain other existing provider kinds; rather than just adding new provider kinds below existing provider kinds in the hierarchy, we can sandwich a new provider kind between existing provider kinds.

As in other descriptive naming schemes, each data provider is assumed to have a set of named properties that can be used for expressing constraints. In contrast to schemes in which a descriptive name consists of a set of attribute-value pairs, tested for equality with provider property values, we allow a descriptive name to include any boolean combination of arbitrary tests on property values. For example, to test whether x and y properties of a data provider specify coordinates within a certain rectangle, we test whether that value of each property is *less than* one specific value and *greater than* another; this constraint cannot be expressed by equality tests. To test whether the x and y properties specify any point within a *pair* of rectangles, we test whether the point lies within the first rectangle, *or* within the second rectangle; this constraint cannot be expressed by listing a set of tests on individual properties, all of which must be satisfied simultaneously. We supplement properties that a boolean constraint may refer to with named *activation parameters* that are required be specified in a descriptive name for a provider of a particular kind. These specific values may be needed to establish a connection with a data provider.

Some descriptive naming systems have a fixed metric associated with named entities for determining which entity a name should resolve to when there are several eligible candidates. We improve upon this approach in two ways. First, we enable an application to specify any numeric combination of property values to be used as a metric. Different descriptive names can specify different metrics. Second, rather than simply using the metric to determine that one data provider will be selected over another, our scheme allows descriptive names for sets of multiple data providers, such as those with the 10 highest metric values or all those with a metric value greater than 50.

Our property tests, application-defined metrics, and activation parameters are closely integrated with our system of provider kinds. The properties that may be named in a boolean constraint or a metric are determined by the provider kind; any property defined for a particular provider kind must be defined for subkinds of that provider kind, thus ensuring that a boolean constraint or metric that can be applied to a provider of a given kind can also be applied to a provider of any subkind of that kind. Similarly, the activation parameters for which values must be supplied in a descriptive name are determined by the provider kind. Any activation parameter required to be specified for providers of a given kind is also required to be specified for providers of all superkinds of that kind.

3. A Programming Model for Context Applications

In our model, context-aware applications obtain data from *data providers*. A data provider has a *current value* that may change from time to time. All data providers respond to requests for their current values. Some data providers are also active, taking the initiative to report that they have generated new values. A web service that responds to requests for the current price of a given stock is a passive data provider. A sensor that issues a report whenever it detects motion is an active data provider.

An application obtains context data from a service with a straightforward interface: The application issues a descriptive name called a *provider query* to the service, and the service responds with a list of one or more handles for context-data providers, registered with the service, that satisfy the query. Each handle has a descriptor reporting distinguishing properties of its data provider. Given a data-provider handle, an application can request the current value of the data provider, or subscribe to receive a notification each time the data provider generates a new value. Thus, the subject of a provider query is not a value, but the continuously evolving stream of values associated with a data provider.

4. The Nature of a Provider Query

We discuss underlying concepts related to provider queries in Sects. 4.1 and 4.2, and turn to provider queries themselves in Sect. 4.3.

4.1. Provider Kinds

Every data provider is registered with the name-resolution service as belonging to some provider kind. The definition of a provider kind specifies the type of the values returned by the provider, the names and types of its activation parameters, and a set of attributes describing properties of the provider. Activation parameters provide the information needed to initialize a data-provider handle. Activation parameters might include, for example, the unique identifier of a particular real-world entity, or an authentication token.

Fig. 1 defines a provider kind for providers that return the location of a vehicle with a specified vehicle identification number (VIN). A provider of kind `VINToLocation` pro-

```
Provider kind VINToLocation:
    Type of provided values: LatLongType
    Activation parameters:
        vehicleID: VINType
    Properties:
        radiusOfErrorInMeters: float
        freshnessInSeconds: int
```

Fig. 1. Definition of provider kind VINToLocation

vides values of type `LatLongType` and is activated with a parameter `vehicleID` of type `VINType`. The definition in Fig. 1 says nothing about the semantic relationship between the `VINType` value used for activation and the `LatLongType` value provided—for example, that the value provided is the location of the vehicle with the specified VIN. The definition could apply just as easily to a kind for providers that give the location of *the registered owner of* the vehicle with the specified VIN. We expect a provider kind to reflect a particular semantic relationship; providers with different semantics belong to different provider kinds, say `VINToVehicleLocation` and `VINToOwnerLocation`, that may happen to have the same provided type, activation parameters, and properties.

We do not attempt to formalize the semantics of a provider kind. Rather, we rely on the humans who name provider kinds in queries to be familiar with the intended semantics of those provider kinds, just as users of a relational database are expected to be familiar with the semantics of the tables and columns they name in SQL queries.

4.1.1. Subkinds and Superkinds. Provider kinds can be organized into hierarchies of superkinds and subkinds, such that a query for a provider of kind *k* can be satisfied by a provider of any subkind of *k*. To formalize this hierarchy, we assume that the types to which provided values and activation-parameter values belong are themselves organized in a supertype-subtype hierarchy. A provider kind *p* is allowed to be the direct parent of a child provider kind *c* only if each of the following conditions holds:

- The type of value provided by *c* is a subtype of the type of value provided by *p*.
- For each activation parameter of kind *c*, kind *p* has an identically named activation parameter, and the type of each parameter of *c* is a supertype of the type of the corresponding parameter of *p*. (Thus the set of parameter values understood by a provider of kind *c* includes *at least* every parameter value understood by a provider of kind *p*; *p* may have "extra" parameters that have no counterpart in *c*, which can be ignored when activating a data provider of kind *c* as if it were of kind *p*.)
- The set of properties of *c* is a superset of the set of properties of *p*.

To these formal conditions, we add an informal one:

- The semantics of *c* (as understood informally by a human) are consistent with the semantics of *p*.

(The formal conditions determine when it is *legal* for *p* to be a direct parent of *c*, and the informal condition determines when it is *appropriate*.) The *superkinds* of a provider kind *k* consist of *k* and the superkinds of all direct parents of *k*; if *x* is a superkind of *y*, then *y* is a *subkind* of *x*. (Every provider kind is a subkind and a superkind of itself.)

For example, suppose the type LatLongType, giving a two-dimensional location in terms of latitude and longitude, has a subtype LatLongElevType, giving a three-dimensional location that also includes elevation. Fig. 2(b) defines a kind for providers of three-dimensional locations of vehicles with a given VIN. Because LatLongElevType is a subtype of LatLongType, VINTo3DLocation can be a subkind of VINToLocation. Then a query for a provider of kind VINToLocation could be satisfied by a provider of kind VINTo3DLocation; an application would use the LatLongElevType values it receives from the provider as if they were LatLongType values. Some providers of vehicle-location information might use GPS receivers, and for those providers it is meaningful to define an additional property, the number of GPS satellites contributing to the reading. Fig. 2(c) defines a provider kind for these GPS-based providers. VINToGPSLocation is also eligible to be a subkind of VINToLocation, since its properties include all the VINToLocation properties. Any query for a provider of kind VINToLocation can be satisfied by a provider of kind VINToGPSLocation. We can also define a provider kind for GPS-based providers of three-dimensional location, as shown in Fig. 2(d). VINToLocation, VINTo3DLocation, and VINToGPSLocation all qualify to be direct parents of VINToGPS3DLocation. If we define VINTo3DLocation and VINToGPSLocation to be direct parents of VINToGPS3Dlocation, we obtain the hierarchy shown in Fig. 3.

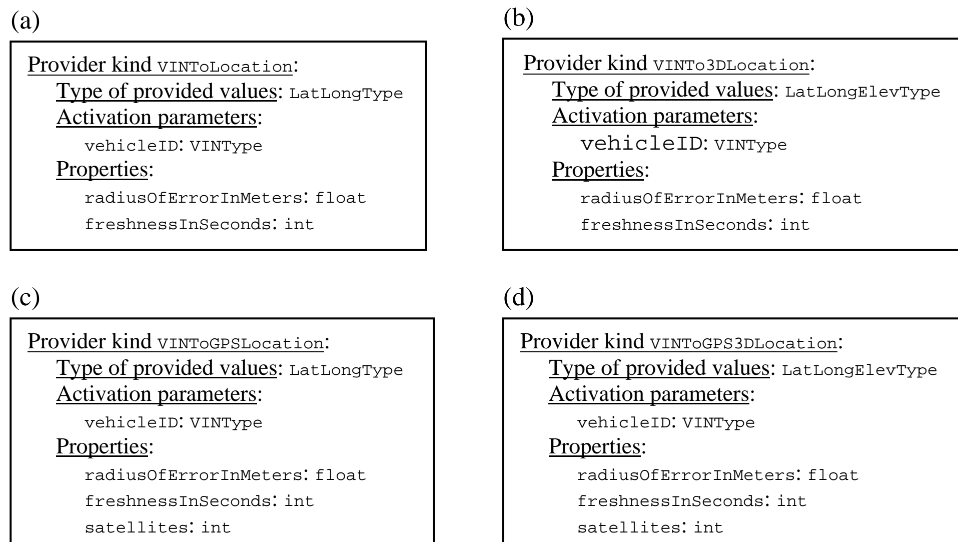

Fig. 2. Definitions of provider kinds (a) VINToLocation (as in Fig. 1), (b) VINTo3DLocation, (c) VINToGPSLocation, and (d) VINToGPS3DLocation

4.1.2. Bottom-Up Definition of Superkinds. Traditionally, subtype hierarchies are built from the top down; that is, the definition of a type names its direct parents, which must

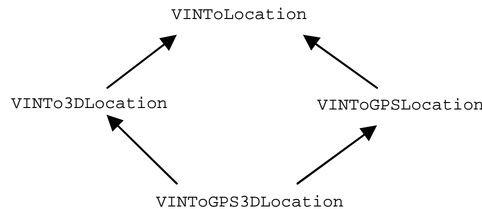

Fig. 3. A multiple-inheritance subkind hierarchy. A query for a provider of a given kind can be satisfied by a provider of that kind or of any kind directly or indirectly below it in the hierarchy.

have been defined earlier. In contrast, the definition of a new provider kind names both direct parents and direct children. Thus the new provider kind can be installed as a superkind of existing kinds, as a subkind of existing kinds, or both.

Just as top-down growth of a hierarchy allows for specialization, bottom-up growth allows for *generalization*. Generalization allows the vocabulary of provider queries to evolve without disruption as new provider kinds are devised. We give two examples.

First, suppose a standard type `TelematicsData` has been extended independently by company X to type `XTelematicsData` and by company Y to type `YTelematicsData`. Each company markets a device that reports a value of its own extended type, given a VIN. The companies define corresponding provider kinds `VINToXTelematicsData` and `VINToYTelematicsData`. We are managing a fleet that had been using X's device to obtain standard `TelematicsData` values (treating `XTelematicsData` values as `TelematicsData` values, ignoring X's extensions). We have now added vehicles with Y's devices to the fleet. So that we can write a query that will find *all* providers of `TelematicsData` values, we define a new provider kind, `VINToTelematicsData`, as a superkind of `VINToXTelematicsData` and `VINToYTelematicsData`.

A second form of generalization involves activation parameters. Suppose we have data providers of kind `VINToLocation`, which provide the location of a vehicle given its VIN, and data providers of kind `PlateToLocation`, which provide the location of a vehicle given its plate number. Suppose we have both the VIN and plate number of all vehicles of interest. Rather than issue one query for `VINToLocation` and, if that fails, a second query for `PlateToLocation`, we can define a new provider kind `VINAndPlateToLocation`, which takes *both* a VIN and a plate number as activation parameters. Then `VINAndPlateToLocation` can be defined as a superkind of `VINToLocation` and `PlateToLocation`. We can issue a single query for `VINAndPlateToLocation` data providers, which will be satisfied by both `VINToLocation` and `PlateToLocation` data providers.

4.2. Provider Descriptors

Every data provider has a *provider descriptor* conveying the identity of the provider and information about its state. This may include static information about the nature and capabilities of the data provider as well as dynamic information such as the provider's current value and the quality of service currently being provided. A provider descriptor includes:

- a unique identifier for the data provider
- the name of its provider kind
- values for the properties defined for providers of that kind
- the provider's current value

If a provider kind *s* is a subkind of a provider kind *k*, a descriptor for a provider of kind *s* includes at least the properties found in a descriptor for a provider of kind *k*.

4.3. Provider Queries

A provider query is a test that a provider descriptor either passes or fails. It includes:

- the name of a provider kind, indicating that a provider of that kind or one of its subkinds is desired
- values for the activation parameters associated with that provider kind
- a predicate, possibly referring to the values of properties associated with the provider kind, to be applied to the property values in a given provider descriptor, indicating whether the descriptor should be considered to satisfy the query
- a *selection mechanism* for determining which provider descriptors, among those determined to satisfy the query, should be returned in the query result

A selection mechanism can have one of the following forms:

- *all*, indicating that the result should contain all provider descriptors satisfying the query
- *first*(*k*), indicating that the result should contain at most the first *k* provider descriptors found that satisfy the query
- *top*(*k*,*expression*), where *expression* is a numeric expression involving properties in the provider descriptor for the specified provider kind, indicating that the result should contain up to *k* provider descriptors for which *expression* has a maximal value
- *ge*(*expression*,*n*), where *expression* is a numeric expression involving properties in the provider descriptor for the specified provider kind, indicating that the result should contain all descriptors for which *expression* has a value greater than or equal to *n*

4.3.1. Predicates Versus Activation Parameters. Predicates and activation parameters play distinct roles. A predicate tests whether properties of a data provider satisfy certain conditions, but need not constrain a property to hold one specific value. Activation parameters supply specific values needed to establish a connection to a data provider.

Sometimes, the same information must be supplied as an activation parameter and in a predicate. Consider a query for providers of IBM stock prices: Some of these providers might be general stock-quote services, which require a stock symbol to be passed as an activation parameter; others might be dedicated specifically to providing the price of IBM stock. Such providers belong, respectively, to provider kind `PriceBySymbol` and its sub-

kind `IBMPrice`, defined in Fig. 4. A query for `PriceBySymbol` can be satisfied by a provider of either kind. However, such a query would also be matched by providers belonging to other subkinds of `PriceBySymbol`, such as `IntelPrice` and `Microsoft-Price`. To filter out these other data providers, we write a query that not only specifies a value of `"IBM"` for `symbolParameter` (as required for providers of kind `PriceBy-Symbol`) but also specifies the predicate `symbolProperty="IBM"`.

Provider kind `PriceBySymbol`:
  Type of provided values: `USDollars`
  Activation parameters:
    `symbolParameter: string`
  Properties:
    `symbolProperty: string`
    `tickerDelayInMinutes: int`

Provider kind `IBMPrice`:
  Type of provided values: `USDollars`
  Activation parameters: (none)
  Properties:
    `symbolProperty: string`
    `tickerDelayInMinutes: int`

Fig. 4. Definition of provider kind PriceBySymbol and its subkind IBMPrice

With descriptive naming schemes that test attributes only for equality with specific values, there is no need for both parameters and properties: The string "`symbol=IBM`" acts both as a specification of the value to be used for `symbol` (when activating a data provider requiring a specific value) and as a test to be performed on the value of the property `symbol` (when filtering provider descriptors that contain a `symbol` property). However, by restricting a query to be, in essence, a conjunction of equalities, such a scheme precludes queries for, say, a stock-price provider with a ticker delay *less than* 20 minutes.

4.3.2. Semantics of a Provider Query. We define the semantics of a provider query operationally: A provider query specifying a provider kind *pk*, activation parameters $ap_1,\ldots,ap_n$, predicate *p*, and selection mechanism *sm* is resolved as if by the following steps:

1. Attempt to activate every data provider registered as belonging to some subkind of *pk*, using the activation-parameter values $ap_1,\ldots,ap_n$.
2. For each successfully activated provider, construct a provider descriptor appropriate for kind *pk* with the properties and current value of that provider.
3. Apply the predicate *p* to each provider descriptor and include all those for which the result is *true* in a set of candidates.
4. Apply the selection mechanism *sm* to select a result set from the set of candidates.

The effect of these steps can often be achieved more efficiently: If the predicate does not refer to dynamic properties of a data provider, it can be applied to an *approximate descriptor*, containing only static properties, created without actually activating the provider. (This approach is reminiscent of the approximate caches of [4].) If the predicate refers to dynamic properties other than the provider's value, it is necessary to activate the provider, but not to retrieve its current value. For a selection mechanism of the form *first*(*k*), the query processing can be stopped after *k* descriptors have been obtained. Traditional indexing and query-optimization techniques can be applied to static properties to avoid the re-

trieval of provider descriptors that cannot possibly satisfy the predicate in a query.

A predicate referring to a data provider's value is potentially costly: In the worst case, resolving a name entails activating every provider that has a suitable provider kind, and requesting its current value. However, this feature is also very powerful: We can query for all vehicles currently located in a specified region, or all sensors currently sensing out-of-range temperatures. Fortunately, a name resolver can be designed so that the cost is borne only by queries that explicitly refer to a provider's value.

5. Prototype Implementation

We have implemented a resolution service for our provider queries. The service is part of the Context Weaver [5] middleware for context-aware applications. Context Weaver collects and combines data from a wide variety of data providers. Context-aware applications running on Context Weaver include one using active-badge data to issue context-aware reminders, one setting priorities for hospital nurses based on data from simulated monitors, and one estimating the availability of individuals based on a variety of context data.

All runtime values in Context Weaver have XML representations, and belong to XML Schema [6] types. Each provider kind is registered as providing values of a particular XML Schema type, and as having activation parameters of particular XML Schema types. XML Schema types can have subtypes; the subtype hierarchy is used to determine whether one given provider kind is allowed to be a direct parent of another.

A Context Weaver provider descriptor is an XML document. XQuery [7], a language that can specify computations on the contents of an XML document, is a natural medium for specifying computations on the values in a provider descriptor. The predicate in a Context Weaver provider query is a boolean XQuery expression. The selection mechanisms *top*(*k*,*expression*) and *ge*(*expression*,*n*) contain numeric XQuery expressions.

Context Weaver is implemented in Java. Registrations of Context Weaver data providers and provider kinds are stored in a relational database accessed through JDBC. Data-provider registrations are indexed by provider kind. We select from the database all provider registrations whose provider kinds are subkinds of the kind specified in the provider query. Our prototype implementation then naively processes provider queries in accordance with the operational definition given in Sect. 4.3.2: An attempt is made to activate each data provider whose registration was retrieved, using the activation-parameter values found in the query. If this attempt is successful, a completely filled-in XML provider descriptor is obtained, and the XQuery predicate in the provider query is applied to this descriptor. If the predicate is true, the provider descriptor is added to a list of candidates. After each retrieved provider registration has been processed in this way, the selection mechanism is applied to the list of candidates to obtain the result of the query. Despite the fact that we have not yet implemented any of the performance improvements envisioned in Sect. 4.3.2, we are able to process over 10 provider queries a second using the IBM Java 1.42 JVM running on a 3GHz Pentium 4 processor with 500MB RAM.

While Context Weaver provides a compelling proof-of-concept for the descriptive naming system proposed in Sect. 4, the focus of this paper is on the naming system itself rather than on any particular implementation of it. Therefore, further details about Context Weaver (such as mechanisms for continually rebinding to the best available providers over the lifetime of a provider query) are beyond the scope of this paper.

6. Related Work

As explained in Sect. 2, two limitations characterize previous approaches to descriptive naming. Some approaches lack expressiveness, effectively restricting the conditions that can be tested to a conjunction of equalities. Some approaches do not support a hierarchical classification of descriptively named entities akin to our provider-kind hierarchy.

The Lightweight Directory Access Protocol (LDAPv3) [8] allows directory-entry attributes to be tested by an arbitrary boolean search filter. However, an LDAP directory hierarchy does not reflect superkind-subkind relationships: A query for an entry of type t is not satisfied by an entry whose type is any subtype of t. In the Service Location Protocol (SLPv2) [9], queries include an LDAPv3 search filter. Every SLP service belongs to some *concrete service type*, and concrete service types may be grouped into *abstract service types*. However, only a two-level hierarchy is supported.

In the Ninja project's Service Discovery Service [10], a service has an XML *service description*, analogous to our provider descriptors. A query is a service description with some of the elements removed, and specifies a conjunction of equalities between corresponding values in the query and the service description. The Intentional Naming System [1] and its follow-on Twine [11] take a similar approach: Both queries and resource descriptions are *attribute-value trees*. A query matches a resource description if and only if each path from the root of the query tree has a corresponding path in the resource description, so a query is effectively a conjunction of equality tests. While [1] contemplates adding ordering comparisons to queries, [11] exploits the fact that a path in a query matches a path in a resource description only if both paths can be hashed to the same value.

In the *directed diffusion* paradigm [2,12], queries are called *interests*. In [2], an interest is expressed as a set of attribute-value pairs, interpreted as a conjunction of equality tests. Every interest includes an event code naming the subject of the query. Event codes roughly correspond to provider-kind names, but with no inherent relationships among the notions they denote. An enhancement using attribute-comparator-value triples instead of attribute-value pairs is presented in [12]; an interest is still equivalent to a conjunction of simple tests, but the simple tests may include ordering comparisons as well as equalities.

The Jini [13] Lookup Service discovers Java objects representing services. A service object is described by a *service item* that includes a *service identifier* and *attribute sets*. A query is a *service template* that may include a service identifier, Java types, and *attribute-set templates*. A service template matches a service item if its service identifier matches the service identifier in the service item, the service object in the service item is an in-

stance of each Java type in the service template, and each attribute-set template in the service template matches an attribute set in the service item. The hierarchy of Java service-object subclasses can play a role analogous to our subkind hierarchy, but the attribute-set template specifies a conjunction of exact matches with specified values.

Universal Description, Discovery, and Integration (UDDI) [14] is a framework for issuing queries to discover web services. The UDDI registry was originally conceived of as a "yellow pages" directory, in which services are looked up by locating a business in a given industry and then examining the services offered by that business. Businesses are categorized by industry according to a hierarchy, but this hierarchy has no semantic role in the processing of queries. Services are not looked up based on their semantic properties.

In contrast to these approaches that are less flexible than ours, ontology-based systems aspire to provide greater flexibility. The OWL Web Ontology Language [15], based on the Resource Description Framework [16], is one notation for defining ontologies. Ontology-based systems aim to support unstructured (e.g., natural-language) queries, and to apply common-sense reasoning to deduce facts that are not explicitly represented. By defining relationships between terms in different vocabularies, an ontology also provides a bridge between the vocabularies of a query and a provider descriptor, allowing providers of "thermometer reading" to be discovered in response to a query for "temperature." The goals of ontology-based systems are ambitious, but their promises are unproven. Development of ontologies is labor-intensive, so few exist yet, and it is not clear that resources will exist in the long run to maintain them. Our hierarchy of provider kinds can be viewed as a kind of primordial ontology, with less ambitious and therefore more attainable goals. We seek to classify only data providers rather than arbitrary knowledge, and we do so in a highly constrained manner.

7. Summary

We have proposed a powerful approach for naming context data providers. Our names *describe* the desired properties of value streams rather than identifying particular data providers. Descriptive naming allows a name resolver to select the best available provider dynamically, isolates client applications from dependence on one particular provider, allows providers to be added to or removed without modifying client applications, and makes applications portable to environments with different sets of resources.

Data providers registered with the name resolver are classified according to a multiple-inheritance hierarchy of provider kinds. New provider kinds can be inserted in this hierarchy not only below, but also above specified existing provider kinds, facilitating the introduction of a new provider kind that generalizes previously existing provider kinds. The current state of a registered data provider is described by a provider descriptor with content that depends on its provider kind. A provider query specifies the name of a provider kind, a set of activation-parameter values meaningful for that provider kind, a predicate applicable to descriptors for providers of that kind, and a selection mechanism specifying

how data providers are to be selected from among those that are eligible. The descriptor includes the current value of a data provider, enabling queries for all data providers currently providing values that satisfy a particular condition.

Our descriptive names, or provider queries, are applicable to arbitrary domains, and to sorts of context-data providers not yet conceived of. At the same time, they are precise and unambiguous. A provider query is amenable to consistency checks to ensure that it refers to only attributes that are meaningful for the kind of data provider it describes. It is possible to perform general tests on attributes, including range tests and disjunctions.

The naming system we have proposed has been implemented in the Context Weaver middleware. However, the true test of our approach can only come over the course of years. The approach should be deemed successful if it supports the incorporation of new, unanticipated provider kinds, and if it supports the precise expression of queries by new, unanticipated applications.

References

1. Mic Bowman, Saumya K. Debray, and Larry L. Peterson. Reasoning about naming systems. ACM Transactions on Programming Languages and Systems 15, No. 5 (November 1993), 795–825

2. Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. Proceedings of the Sixth Annual International Conference on Mobile Computing and Networking (MobiCom 2000), Boston, Massachusetts, August 6–11, 2000, 56–67

3. 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

4. Chris Olston, Boon Thau Loo, and Jennifer Widom. Adaptive precision setting for cached approximate values. Proceedings of the ACM SIGMOD International Conference on Management of Data, Santa Barbara, California, May 21–24, 2001, 355–366

5. Norman H. Cohen, James Black, Paul Castro, Maria Ebling, Barry Leiba, Archan Misra, and Wolfgang Segmuller. Building context-aware applications with Context Weaver. IBM Research Report RC 23388, October 22, 2004

6. David C. Fallside, ed. XML Schema Part 0: Primer. W3C Recommendation, May 2, 2001 <URL: `http://www.w3.org/TR/xmlschema-0/`>

7. Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, Jérôme Siméon. XQuery 1.0: An XML Query Language. W3C Working Draft, May 2, 2003 <URL: `http://www.w3.org/TR/xquery/`>

8. M. Wahl, T. Howes, and S. Kille. Lightweight Directory Access Protocol (v3). IETF RFC 2251, December 1997 <URL: `http://www.ietf.org/rfc/rfc2251.txt` >

9. E. Guttman, C. Perkins, J. Veizades, M. Day. Service Location Protocol, Version 2. IETF RFC 2608, June 1999 <URL: `http://www.ietf.org/rfc/rfc2608.txt`>

10. Steven E. Czerwinski, Ben Y. Zhao, Todd D. Hodes, Anthony D. Joseph, and Randy H. Katz. An architecture for a secure service discovery service. Proceedings of the Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '99), Seattle, Washington, August 15–19, 1999, 24–35

11. Magdalena Balazinska, Hari Balakrishnan, and David Karger. INS/Twine: a scalable peer-to-peer architecture for intentional resource discovery. International Conference on Pervasive Computing (Pervasive 2002), Zurich, Switzerland, August 26–28, 2002, 195–210

12. John Heidemann, Fabio Silva, Chalermek Intanagonwiwat, Ramesh Govindan, Deborah Estrin, and Deepak Ganesan. Building efficient wireless sensor networks with low-level naming. Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP 2001), Banff, Alberta, October 21–24, 2001, 146–159

13. Sun Microsystems. Jini Technology Core Platform Specification. Version 2.0, June 2003 <URL: http://wwws.sun.com/software/jini/specs/>

14. Tom Bellwood, Luc Clément, Claus von Riegen, eds. UDDI version 3.0.1. UDDI Spec Technical Committee Specification, October 14, 2003 <URL: http://uddi.org/pubs/uddi_v3.htm>

15. Deborah L. McGuinness and Frank van Harmelen, eds. OWL Web Ontology Language overview. W3C Candidate Recommendation, August 18, 2003 <URL: http://www.w3.org/TR/owl-features/ >

16. Frank Manola and Eric Miller, eds. RDF Primer. W3C Working Draft, October 10, 2003 <URL: http://www.w3.org/TR/rdf-primer/>

Acknowledgements

The Context Weaver system and the context-aware applications described in Sect. 5 were developed by a team of IBM researchers. At various times, this team has included, in addition to the authors, James Black, Marion Blount, John Davis, Maria Ebling, Qi Han, Srikant Jalan, Hui Lei, Barry Leiba, Apratim Purakayastha, Wolfgang Segmuller, and Daby Sow.