Enabling Applications in Sensor-based Pervasive Environments

Nanyan Jiang, Cristina Schmidt, Vincent Matossian and Manish Parashar
The Applied Software Systems Laboratory
Department of Electrical and Computer Engineering
Rutgers University, Piscataway NJ 08855, USA
{*nanyanj, cristins, vincentm, parashar*}*@caip.rutgers.edu* *

Abstract

Supporting new emerging applications in broadband sensor-based pervasive environments requires a programming and management paradigm where the behaviors as well as the interactions of application elements (sensors, actuators, services, etc.) are dynamic, opportunistic, and context, content and capability aware. In this paper we present a programming model that enables opportunistic application flows in pervasive environments. The model builds on content-based discovery and routing services and defines associative rendezvous (AR) as an abstraction for content-based decoupled interactions. Cascading local behaviors (CLB) then build on associative rendezvous to enable opportunistic application flows to emerge as a result of context and content based local behaviors. In this paper we also present the design, prototype implementation and experimental evaluation of the Meteor programming framework and content-based middleware.

1 Introduction

The growing ubiquity of sophisticated and heterogeneous wired and wireless broadband sensor/actuator devices with embedded computing and communications capabilities [7], and the emergence of pervasive information and computational Grid are enabling new generations of applications based on seamless access, aggregation, and interactions [17]. For example, one can conceive of a fire management application where computational models use streaming information from video broadband sensors embedded in the building along with real time and predicted weather information (temperature, wind speed and direction, humidity) and archived history data to predict the spread of the fire

and to guide firefighters, warning of potential threats (blowback if a door is opened) and indicating most effective options. This information can also be used to control actuators in the building to manage the fire and reduce damage.

Other examples include scientific/engineering applications that symbiotically and opportunistically combine computations, experiments, observations, and real-time telemetry data to manage and optimize its objectives (e.g. oil production, weather prediction) [17], pervasive applications that leverage the pervasive/ubiquitous information to continuously monitor, manage, adapt, and optimize our living context (e.g. your clock estimates drive time to your next appointment based on current traffic/weather and warns you appropriately), crisis management applications that use pervasive conventional and unconventional information for crisis prevention and response, medical applications that use in-vivo and in-vitro multimedia broadband sensors and actuators for patient management, active monitoring systems for automated highway systems, manufacturing system or unmanned airborne vehicles, and business applications that use anytime-anywhere information access to optimize profits.

The defining characteristic of these applications is their ability to leverage the pervasive infrastructure [13] to continuously manage, adapt, and optimize themselves to meet their objectives. However, the underlying pervasive environments are large, heterogeneous, ad hoc, highly dynamic, and unreliable. This inherent complexity and uncertainty breaks down application development paradigms based on static behaviors and pre-orchestrated compositions. As a result, supporting these applications requires an application programming and management paradigm where the behaviors as well as the interactions of applications elements (sensors, actuators, services, etc.) are dynamic and context, content and capability aware. Further, these interactions and the resulting flows should be opportunistic.

In this paper we present a programming model that enables such opportunistic flows in pervasive environments. The model builds on content-based discovery and routing

services and defines *associative rendezvous* as an abstraction for content-based decoupled interactions. *Cascading local behaviors* then build on associative rendezvous to enable opportunistic application flows where flows emerge as a result of context and content based reactive behaviors.

Associative Rendezvous (AR) [1] is a paradigm for content-based decoupled interactions. It extends the conventional name/identifier-based rendezvous [8, 21] in two ways. First it uses flexible combinations of keywords (i.e. keywords, partial keywords, wildcards, ranges) from a semantic information space, instead of opaque identifiers that have to be globally synchronized. Second, it enables the reactive behaviors at rendezvous points to be embedded in the message or message request. AR differs from emerging publish/subscribe paradigms in that individual interests (subscriptions) are not used for routing and do not have to be synchronized - they can be locally modified at a rendezvous node at anytime. In the Cascading Local Behavior (CLB) programming model, the behaviors of individual application elements (i.e., sensors, actuators, services) are locally defined in terms of local state, and context and content events, and result in data and interest messages being produced. Interactions, compositions and application flows emerge as a consequence of the cascading effect of such local behaviors, without having to be explicitly programmed.

In this paper we also present the design, prototype implementation and experimental evaluation of Meteor, a programming framework and content-based middleware that support the programming model and provide a flexible platform for the evaluation of future sensor/actuator-based wired/wireless systems.

The rest of this paper is organized as follows. Section 2 outlines the AR interaction paradigm. Section 3 defines the CLB programming model with an illustrative example. Section 4 introduces the programming framework and the overall architecture of Meteor and describes its design and implementation. Section 5 presents the evaluation results. An overview of related work is presented in Section 6. The paper concludes with Section 7.

2   Associative Rendezvous (AR)

Associative Rendezvous (AR) is a paradigm for content-based decoupled interactions with programmable reactive behaviors. Rendezvous-based interactions [8] provide a mechanism for decoupling senders and receivers. *Rendezvous point* (*RP*), the place where rendezvous interactions occur, may be a broadband access point, a forwarding node in a sensor network or a server or a peer node in a wired network. Senders send messages to a *rendezvous point* without knowledge of who or where the receivers are. Similarly,

---

[1] The term associative is used to refer to messaging based on content rather than addresses [2].

receivers receive messages from a *rendezvous point* without knowledge of who or where the senders are. Note that senders and receivers may be decoupled in both space and time [8]. Such decoupled asynchronous interactions are naturally suited for large, distributed and highly dynamic systems such as pervasive environments.

In conventional rendezvous interactions, *rendezvous points* are defined by opaque identifiers [21] that have to be globally synchronized before they can be used. This limits both the scalability and the dynamism of the system. Associative interactions [2, 3] use semantic content-based resolution, used by the naming service, to enable interactions. In associative interactions participating clients locally maintain and export "profiles" consisting of attributes specifying credentials, context, state, interests, and capabilities. Messages are similarly enhanced to include "semantic-selectors". The semantic-selector is a prepositional expression over all possible attributes and specifies the profile(s) that are to receive the message. Thus the notion of a static client or client group name used by conventional interactions is subsumed by the selector which descriptively names dynamic sets of clients of arbitrary cardinality. Associative interactions only require the existence of globally known information spaces (ontologies), and eliminates the need for expensive synchronization and complex tracking protocols in pervasive environments.

AR extends the conventional name/identifier-based rendezvous in two ways. First, it uses flexible combinations of keywords (i.e, keyword, partial keyword, wildcards, ranges) from a semantic information space, instead of opaque identifiers (names, addresses) that have to be globally known. Interactions are based on content described by keywords, such as the type of data a sensor produces (temperature or humidity) and/or its location, the type of functionality a service provides and/or its QoS guarantees, and the capability and/or the cost of a resource. Second, it enables the reactive behaviors at the rendezvous points to be encapsulated within messages increasing flexibility and expressibility, and enabling multiple interaction semantics (e.g. broadcast, multicast, notification, publish/subscribe, mobility, etc.).

### 2.1   The Semantics of Associative Rendezvous Interactions

The AR interaction model consists of three elements: *Messages*, *Associative Selection*, and *Reactive Behaviors*.

AR Messages: An AR message is defined as the triplet: (*header*, *action*, *data*). The data field may be empty or may contain the message payload. The header includes a semantic *profile* in addition to the credentials of the sender, a message context and the TTL (time-to-live) of the message. The profile is a set of attributes and/or attribute-value pairs, and defines the recipients of the message. The at-

Actions	Semantics
store	store data profile and data; match message profile with existing interest profile; execute action if match.
retrieve	match message profile with existing data profiles; send data corresponding to each matching data profile to the sender.
notify_data notify_interest	match message profile with existing data/interest profiles; notify sender if there is at least one match.
delete_data delete_interest	match message profile with existing data/interest profiles; remove all matching data profiles and data from the system in case of delete_data; remove all matching interest profiles from the system in case of delete_interest.

Table 1. Basic reactive behaviors.

tribute fields must be keywords from a defined information space while the value field may be a keyword, partial keyword, wildcard, or range from the same space. At the rendezvous point, a profile is classified as a *data profile* or an *interest profile* depending on the action field of the message. A sample data profile used by a sensor to publish data

(a)   (b)
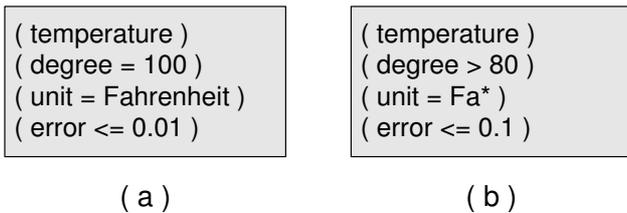
Figure 1. Sample message profiles: (a) a data profile for a sensor; (b) an interest profile for a client.

is shown in Figure 1 (a), and a matching interest profile is shown in Figure 1 (b). Note that the number or order of the attribute/attribute-value pairs in a profile are not restricted. However our current prototype requires that the maximum possible number of attribute/attribute-value pairs must be predefined. The *action* field of the AR message defines the reactive behavior at the rendezvous point and is described later in this section.
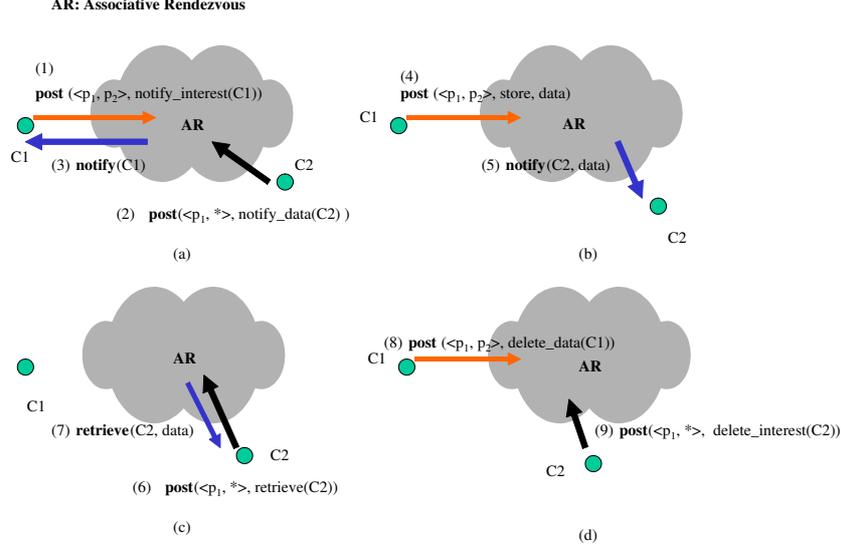
The AR interaction model defines a single symmetric *post* primitive. To send a message, the sender composes a message by appropriately defining the header, action and data fields, and invokes the post primitive. The post primitive resolves the profile of the message and delivers the message to relevant rendezvous points. The profile resolution guarantees that all rendezvous points that match the profile will be identified. However, the actual delivery relies on existing transport protocols. A receive operation is similar except that the action field is defined appropriately and the

data field is empty.

Associative Selection: Profiles are represented using a hierarchical schema that can be efficiently stored and evaluated by the selection engine [6, 2] at runtime. A profile p represents a path in the hierarchical schema, $[e_0 \triangle ...e_k]$, where e_i is an element operand and $\triangle$ can be a parent-child ("/") operator (i.e. at adjacent levels) or an ancestor-descendant ("//") operator (i.e. separated by more than one level). Within a level, the profile defines a propositional expression where $\triangle$ represents propositional operators, such as $\wedge$ and $\vee$ between elements at the same level. Note that the propositional expression at a level must evaluate to TRUE for the evaluation to continue to the next level. The elements of the profile can be an attribute, $e_i : (a_i)$, or an attribute-value pair $e_i : (a_i, v_i)$, where a_i is a keyword and v_i may be a keyword, partial keyword, wildcard or range. The singleton attribute a_i evaluates to true if and only if p contains the simple attribute a_i. The attribute-value pair (a_i, u_i) evaluates to true with respect to a profile p, if and only if p contains an attribute a_i and the corresponding value v_i satisfies u_i, e.g. $v_i = computer$ and $u_i = comp*$. For example, the profile (a) is associatively selected by the profile (b) in Figure 1, since (1) both have matched singleton attribute *temperature*, (2) for attribute *degree*, $100 > 80$, which satisfies the binary relation, (3) for attribute *unit*, *Fahrenheit* matches wildcard *Fa∗*, (4) *error<0.01* satisfies the request *error<0.1*.

A key characteristic of the selection process is that it does not differentiate between interest and data profiles. This allows all messages to be symmetric where data profiles can trigger the reactive behaviors of interest messages and vice versa. The matching system combines selective information dissemination with reactive behaviors. Further, both data and interest messages are persistent with their persistence defined by the TTL field.

Reactive Behaviors: The *action* field of the message defines the reactive behavior at the rendezvous point. Basic reactive behaviors currently defined include *store*, *re-*

AR: Associative Rendezvous

(1)
post (<p_1, p_2>, notify_interest(C1))

AR

C1
(3) **notify**(C1)

C2

(2) **post**(<p_1, *>, notify_data(C2))

(a)

(4)
post (<p_1, p_2>, store, data)

C1

AR

(5) **notify**(C2, data)

C2

(b)

C1

AR

(7) **retrieve**(C2, data)

C2

(6) **post**(<p_1, *>, retrieve(C2))

(c)

(8) **post** (<p_1, p_2>, delete_data(C1))

C1

AR

(9) **post**(<p_1, *>, delete_interest(C2))

C2
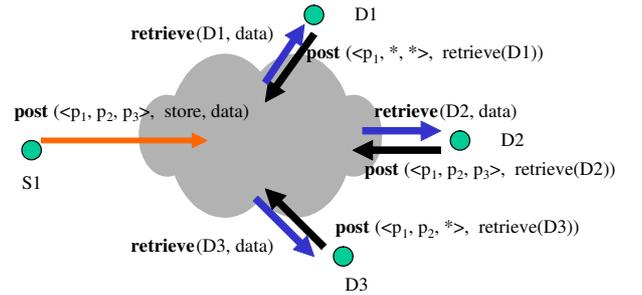
(d)

Figure 2. An illustrative example.

trieve, *notify*, and *delete* as shown in Table 1. The *notify* and *delete* actions are explicitly invoked on a data or an interest profile. The *store* action stores data and data profile at the rendezvous point. It also causes the message profile to be matched against existing interest profiles and associated actions to be executed in case of a positive match. The *retrieve* action retrieves data corresponding to each matching data profile. The *notify* action matches the message profile against existing interest/data profile, and notifies the sender if there is at least one positive match. Finally, the *delete* action deletes all matching interest/data profiles. Note that the actions will only be executed if the message header contains an appropriate credential. Also note that each message is stored at the rendezvous for a period corresponding to the TTL defined in its header. In case of multiple matches, the profiles matching are processed in random order. By default, all matched profiles are returned. However, the programmable reactions can be used to define other behaviors: for instance, *any one* of the matched profile is returned.

2.2   Illustrative Examples

The operation of the model is illustrated in Figure 2. In Figure 2(a), client C1 first requests notification of interest in its data with profile $< p1, p2 >$. Client C2 then requests notification of data corresponding to its interest profile $< p1, * >$. This causes a notification to be sent to C1. C1 now posts data with data profile $< p1, p2 >$ (Figure 2(b)). Since this data profile matches C2's interest profile, a notification is sent to C2. C2 now requests data with interest profile $< p1, * >$ (Figure 2(c)). This matches C1's data profile and the corresponding data is sent to C2. The

example assumes that the TTL for data and interest profiles have not expired. C1 and C2 now delete the data and interest respectively (Figure 2(d)).

As seen in Figure 2(a), a client can subscribe to both interests and data. The use of the *notify* action with the symmetric behavior of the *post* operator allows sensors to save energy by not publishing their data if there is no interest for it. This is particularly important for broadband sensors publishing video/audio data, since the production and transmission of such data may consume a large amount of power and it may be more efficient to produce broadband data only when necessary, for example, there is an interest for that data.

D1
retrieve(D1, data)
post (<p_1, *, *>, retrieve(D1))

post (<p_1, p_2, p_3>, store, data)

retrieve(D2, data)
D2
post (<p_1, p_2, p_3>, retrieve(D2))

S1

post (<p_1, p_2, *>, retrieve(D3))

retrieve(D3, data)
D3

Figure 3. One-to-many interactions using Associative Rendezvous.

AR can be used to realize different interaction semantics such as one-to-many, one-to-all while appropriately setting data and interest profiles. Further, as these profiles are de-
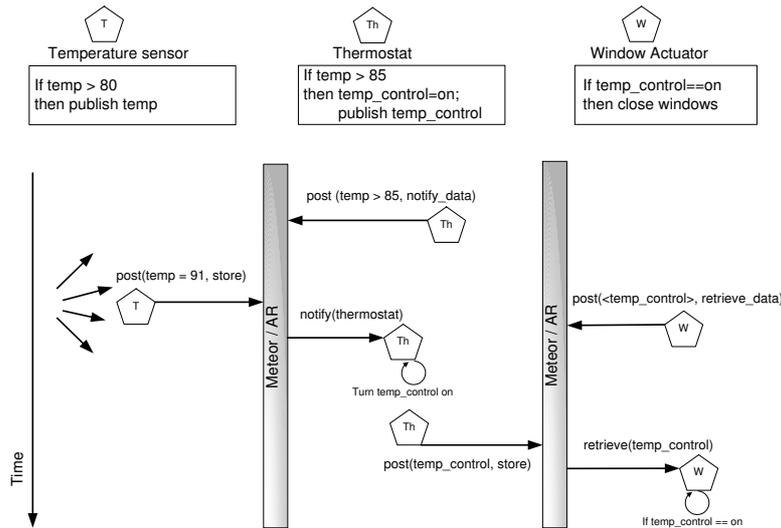
Figure 4. Cascading local behaviors - an illustrative example.

fined locally by a client, no synchronization is required to achieve these interaction semantics. Figure 3 illustrates a one-to-many (e.g. multicast) interaction using AR.

3   Cascading Local Behaviors

Cascading Local Behaviors (CLB) is a model for realizing opportunistic application flows in pervasive sensor/actuator-based environments. It builds on a common semantic basis (i.e., ontology and taxonomy) for describing content and context. In the CLB model only the local behaviors for each element (i.e., sensor, actuator, resources, services) are programmed. These behaviors can be viewed as a state machine where the local state is defined in terms of local actions (A), active interfaces (I) and active data and/or interest messages (M), as illustrated in Figure 5. Local state transitions are triggered by context
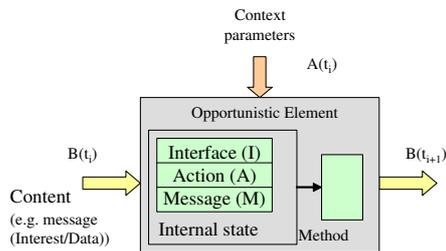

Figure 5. Defining local behaviors.

and/or content events and may result in local actions (e.g., update database or turn on an indicator) and the generation of data/interest messages. Note that the definition of the lo-

cal behavior is independent of the rest of the pervasive system. Messages generated during local state transitions may trigger transitions in other elements, which in turn may generate further messages. The resulting cascading local transitions cause application flows to emerge opportunistically.

For example, consider a very simple smart home scenario with the sensors and actuators shown in Figure 4. The local behaviors of these sensors/acutators are illustrated as `if-then` rules in the figure. The temperature sensor monitors local temperature and generates a *post(temp = 91, store)* data message when the temperature rises above some threshold. This causes AR to generate a notification to the thermostat actuator which then turns the air conditioning on and generates a *temp-control* message. Other sensors/actuators that are its subscribers will be notified and react, for example, a window could shut itself or a fan could turn itself off. The devices need not know each other as long as they use a common semantic basis for describing content and context.

Note that CLB differs from traditional composition-based programming approaches, where the desired end-to-end behavior is known a priori and is used to define the local behaviors of the elements as well as their interactions. In CLB, the behaviors of elements can be independently defined without knowledge of the functionality or existence of other elements. Further, elements in the system can be spatially and temporally decoupled, i.e., an element entering the system at a later time (for example, the window actuator in Figure 4) can still be part of an emerging flow. A key requirement for CLB is a communication/interaction infrastructure that: (a) is scalable and self-managing, (b) is based on content rather than names and/or addresses, (c) sup-

ports asynchronous and decoupled interactions rather than forcing synchronizations, and (d) provides some interaction guarantees. Such an abstraction is provided by AR.

4 Meteor Programming Framework and Content-based Middleware

A schematic overview of the Meteor programming framework and middleware stack is shown in Figure 6. It builds on a self-organizing overlay that interconnects sensor/actutor services, resources, and data on the information/computational Grid, and implements content-based discovery and routing services. It then implements the AR abstraction for content-based decoupled interactions. Finally, cascading local behaviors provide a model for developing opportunistic applications where application flows emerge as a result of context and content based reactive behaviors.
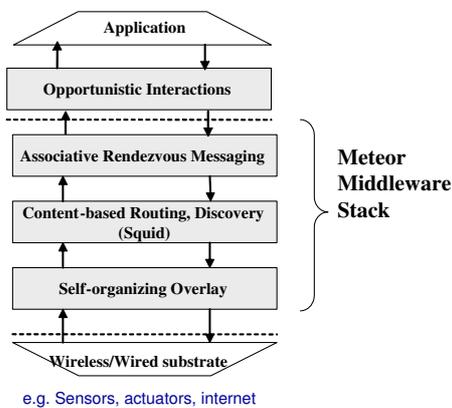

Figure 6. Meteor system architecture

Meteor is part of an NSF funded wireless testbed intended to provide a flexible platform for the evaluation of future sensor/actuator-based systems, wireless networking protocols, middleware, and applications. The testbed is an open-access multi-user experimental research facility consisting of a two-tier architecture involving a controlled laboratory emulator consisting of a radio grid of roughly 400 radio nodes arranged in a regular rectangular grid, and an outdoor field trial network involving an IP-based 3G base-station and related mobile multimedia platforms. The sensor network is a self-organizing, three tiered (hierarchical) structure consisting of low-power sensor nodes with limited functionality, higher-power radio forwarding nodes that route data, and access points that route data between radio links and other wired/wireless infrastructures. The overlay network connects forwarding nodes and access points with the wired infrastructure and supports the Meteor programming framework and content-based middleware. These

components are described below.

Overlay Network: The Meteor overlay network is composed of RP nodes, which may be access points or message forwarding nodes in ad-hoc sensor networks and servers or peer nodes in wired networks. RP nodes can join or leave the network at any time. The current Meteor overlay network uses Chord [22] protocol to organize peers in a ring topology.

The overlay network layer of the middleware stack provides a simple abstraction to the upper layers, consisting of a single operation: **lookup**(*identifier*). Given an identifier, this operation locates the node that is responsible for it, i.e, the node with an identifier that is the closest identifier greater than or equal to the queried identifier. Application names can be mapped to identifiers using hashing mechanisms, and then mapped to nodes in the overlay network.

Content-based Discovery and Routing (Squid): Content-based discovery and routing services in Meteor are provided by Squid [19, 20]. Squid builds on top of the overlay to enable flexible content-based routing. As mentioned above, the **lookup** operator provided by the overlay requires an exact identifer. Squid effectively maps complex queries consisting of keyword tuples (multiple keywords, partial keywords, wildcards, and ranges) onto clusters of identifiers, and guarantees that all peers responsible for identifiers in these clusters will be found with bounded costs in terms of number of messages and the number of intermediate RP nodes involved.

Associative Rendezvous Messaging Substrate (ARMS): The ARMS layer implements the Associative Rendezvous interaction model. At each RP, ARMS consists of two components: the *profile manager* and the *matching engine*. The matching engine component is essentially responsible for matching profiles. An incoming message profile is matched against existing interest and/or data profiles depending on the desired reactive behavior. If the result of the match is positive, then the action field of the incoming message is executed first followed by the evaluation of the action field in matched profiles. The profile manager manages locally stored profiles, and monitors message credentials and contexts to ensure that related constraints are satisfied. For example, a client cannot retrieve or delete data that it is not authorized to. The profile manager is also responsible for garbage collection. It maintains a local timer and purges interest and data profiles when their TTL fields have expired. Finally, the profile manager executes the action corresponding to a positive match.

4.1 Implementation Overview

The current implementation of Meteor builds on Project JXTA (http://www.jxta.org), a general-purpose peer-to-peer framework that provides a set of open protocols and plat-

forms to build new services and applications. JXTA defines concepts, protocols, and a network architecture. JXTA concepts include peers, peergroups, advertisements, modules, pipes, rendezvous and security. JXTA defines protocols for (1) discovering peers (Peer Discovery Protocol, PDP), (2) binding virtual end-to-end communication channels between peers (Pipe Binding Protocol, PBP), (3) resolving queries (Peer Resolver Protocol, PRP), (4) obtaining information on a particular peer, such as its available memory or CPU load (Peer Information Protocol, PIP) (5) propagating messages in a peergroup (Rendezvous protocol, RVP), (6) determining and routing from a source to a destination using available transmission protocols (Endpoint Routing Protocol, ERP). The JXTA architecture builds on three layers, a core layer, for essential common functionalities, a service layer, for additional pluggable/unpluggable behaviors, and an application layer for end-to-end high-level control. Note that JXTA is an application level technology and does not introduce any limitations on the underlying infrastructure or the routing protocols.

The overlay network, Squid and the ARMS layers of the Meteor stack are implemented as event-driven JXTA services. Each layer registers itself as a listener for specific messages, and gets notified when a corresponding event is raised. Since Meteor is designed as an overlay network of rendezvous peers, it is incrementally deployable. A joining RP uses the Chord protocol and becomes responsible for an interval in the identifier space. In this way, the addition of a new rendezvous node is transparent to the end-hosts.

The overall operation of the Meteor overlay consists of two phases: bootstrap and running. During the bootstrap phase (or join phase) messages are exchanged between a joining RP and the rest of the group. During this phase, the RP attempts to discover an already existing RP in the system and construct its routing table. The joining RP sends a discovery message to the group. If the message remains unanswered after a duration (in the order of seconds), the RP assumes that it is the first in the system. If a RP responds to the message, the joining RP queries this bootstrapping RP according to the Chord join protocol and updates routing tables to reflect the join.

The running phase consists of a stabilization and a user mode. In stabilization mode, a RP responds to queries issued by other RPs in the system. The purpose of the stabilization mode is to ensure routing tables are up to date, and to verify that other RPs in the system have not failed or left the system. In user mode, RPs participates in interactions as part of the Squid and ARMS layers. The ARMS matching engine at each RP is based on MySQL (http://www.mysql.com), a lightweight SQL database.

5 Experimental Evaluation

Meteor was experimentally evaluated using a prototype deployment on the Linux-based sensor-network emulation testbed. In the experiments 64 nodes acted as a RP and ran the complete Meteor stack. Profiles at each RP were locally stored in a MySQL database. The overheads at each layer of the stack were measured. Further, we used simulations of up to 5000 RPs and 10^6 unique profiles to evaluate the scalability of the content-based routine layer. The experiments are presented below.
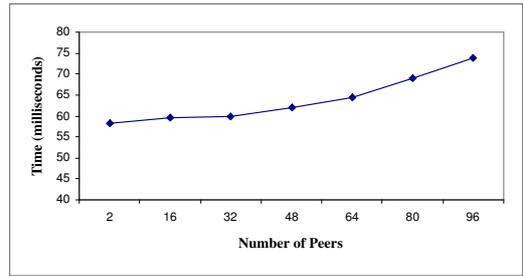
(a)
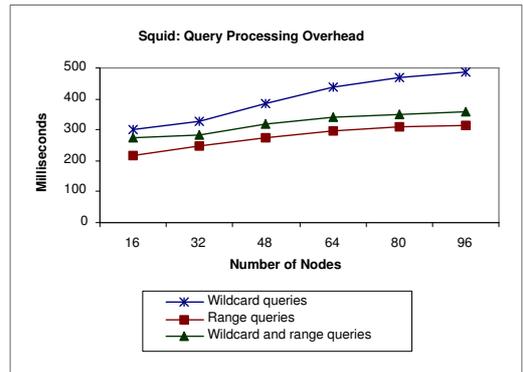
(b)
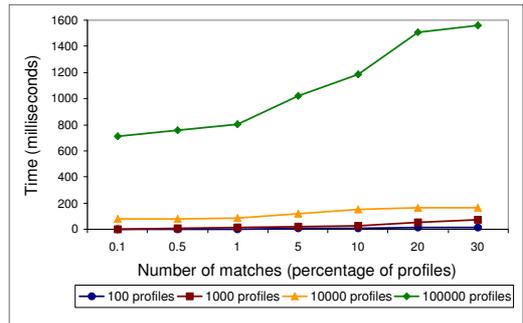
(c)

Figure 7. (a) Overlay network lookup overhead (Chord); (b) Content-based routing overhead (Squid); (c) Matching overhead at a single RP (ARMS).

7

5.1 Overlay Network Layer

This experiment measured the latency for peer lookup in the overlay network as a function of the size of the system. To get an accurate measurement of the latencies, a single peer was run on each node and each peer sent messages to a randomly selected destination peer. Each message required an overlay lookup operation. Average times are plotted in Figure 7 (a). The graph shows that the average elapsed time is not affected by the linear growth of the size of the system, validating the scalability of the overlay network lookup operation and the Chord routing algorithm.

5.2 Content-based Routing Layer

This experiment measured the overhead of routing using complex keyword tuples. Three sets of keyword tuples were used, the first containing wildcards, the second containing ranges, and the third containing both wildcards and ranges. The routing overheads at the Squid layer were measured at each RP and averaged. The results are plotted in Figure 7 (b). The measured overhead includes times for cluster refinments and subcluster lookup. As Figure 7 (b) shows, the overhead grows slowly and at a much smaller rate than the system size. This demonstrates that Squid can effectively scale to large numbers of peers while maintaining acceptable routing times. As expected, the routing times are high for queries with wildcards as they involve a larger number of clusters and correspondingly larger number of peers.

To evaluate the scalability of the content-based routing layer we simulated Squid with up to 5000 peers where each peer is an RP, and up to 10^6 unique profiles. In the simulations, the number of profiles stored in the system increases with the number of RP peers. We used complex keyword tuples to test the routing scalability: keyword tuples with partial keywords and wildcards, and keyword tuples containing ranges. Figure 8 shows results for a 3D keyword space. The number of nodes that process the query is a small fraction of the total nodes, and it increases at a slower rate than the size of the system. For wildcard queries, the average number of processing nodes is below 11%, and the number of nodes that found matches is below 6%. These percentages decrease as the system size increases, demonstrating the scalability of the system. As Figure 8 shows, range queries are more efficient than wildcard queries, which is expected, as query optimization and pruning are more effective for range queries. Additional simulation results can be found in [19].

5.3 Associative Rendezvous Messaging Substrate

This experiment measured the matching overhead at each RP node. The action type for all messages in this experiment was 'notification'. Only the overhead of querying
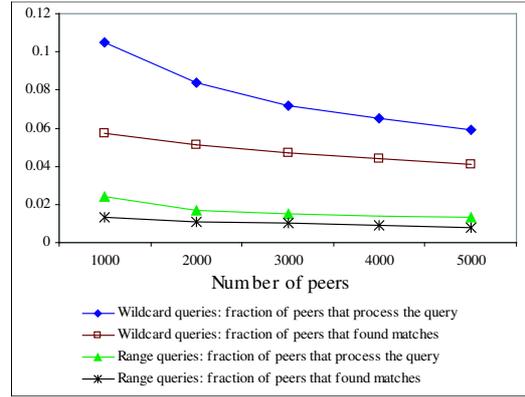
Figure 8. Simulated results for a P2P system with up to 5000 RPs and up to 10^6 interest profiles. The results show the percentage of peers that process a query (on average) and the percentage of peers that found matches, for wildcard queries and for range queries respectively.

the database and of constructing the notification message was considered. The notification delivery was done outside of the Meteor stack and was not measured. The experiment was conducted for profiles containing sets of complex keyword tuples containing wildcards and/or ranges. The number of profiles in the database was varied. The results are plotted in Figure 7 (c). The results were grouped based on the matching expressed as a percentage of the total number of stored profiles, and averaged. As seen in Figure 7 (c), for a moderate-size database (up to 10^4 profiles) the overhead incurred is very low. The overhead increases substantially when 10^5 profiles are stored locally, as could have been expected given the memory and data access times required by such a large number of items. However, we believe that even 10^4 profiles seems greater than needed for an RP.

6 Related Work

Content-based Interactions Content based decoupled interactions has been addressed by publish-subscribe-notify (PSN) models [8]. PSN based systems include Sienna [4] and Gryphon [12]. The associative rendezvous model differs from PSN systems in that individual interests (subscriptions) are not used for routing and do not have to be synchronized - they can be locally modified at a rendezvous node at anytime. While PSN systems provide flexible matching capabilities, scalability remains a concern in these systems.

i3 [21] provides a similar rendezvous-based abstraction and has influenced this work. However, an i3 identifier

is opaque and must be globally known. Associative rendezvous uses semantic identifiers that are more expressive and only require the existence of agreed on information spaces (ontologies). Besides, its dynamic binding semantics enables profiles to be added, deleted or changed on-the-fly.

The associative broadcast [2] paradigm has also influenced this effort. The key difference between this model and associative rendezvous is that the binding of profiles takes place at intermediate nodes instead of the broadcast medium. As a result, associative broadcast only supports transient interactions. Further, its scalability over wide areas is a concern.

The rendezvous-based communication is conceptually similar to tuple space research in distributed systems [18, 14, 24]. A tuple space is a shared space that can be associatively accessed by all nodes in the system. While tuple space is a powerful model for interactions and coordination, efficient and large-scale implementations of pure tuple space based systems is a challenge. Associative rendezvous maintains the conceptual expressiveness of tuple spaces while providing an implementation model that is scalable.

Unlike other rendezvous-based models [10], associative rendezvous enables programmable reactive behaviors at rendezvous points using the action field within a message. In addition, associative rendezvous is able to realize a variety of basic communication services without the need for mobile code [23], or any heavy duty protocols. Further, interactions in the associative rendezvous model are symmetric allowing participants to simultaneously be information producers and consumers. Finally, Grid messaging systems such as the NaradaBrokering [9] focus on persistence and reliable message delivery rather than content-based interactions.

Programming Models for Sensor-based Pervasive Systems Programming models and frameworks for sensor/actuator-based systems are receiving increasing attention [7]. Early efforts included low-level approaches where applications mechanisms are hard-coded into the devices, as well as intermediate level approaches where specialized device operating systems (e.g. TinyOS [15]) or languages (e.g. Nesc [11]), are defined to support application development. The latter approach has the advantages of modularity, multi-tasking, and hardware abstraction. However, in both these approaches, the developer has to create a single executable image that is downloaded into each node. Such node-based programming approaches run counter to the promise of large-scale sensor systems.

More recently, systems such as the Berkeley TinyDB [16], and Cornell Cougar [25], provide higher level programming abstraction that view the sensor network as a distributed database, and enable users to execute queries expressed in a suitable relational language. Optimizations

techniques such as aggregation trees are used to resolve queries efficiently. EnviroTrack [1] is a related approach that adopts a form of attribute-based naming, i.e., context labels, as an addressing scheme to facilitate sensor-based tracking applications. While these techniques are effective for sensor-based monitoring applications, they fall apart for more complex tasks involving collaborative signal processing (as opposed to computation expressed over relations), computational loops, and event processing. This is because they primarily focus on the expressiveness of the task, and not on how the computation to perform that task via in-network processing, which should be structured. In contrast, the AR model using programmable reactions can achieve complex tasks, such as aggregation and query processing.

The TelegraphCQ project [5], which is closest to our approach, views data as fluids for emerging applications and dataflow processing must monitor and react to streams of information as they pass through the network. The primary characteristic of TelegraphCQ is its usage of window-based query semantics for continuous queries. Thus, an efficient filtering mechanism that corresponds to the desired end-to-end application behavior is the basis of TelegraphCQ. CLB differs from TelegraphCQ in that it focuses on the programming of local behaviors rather than requiring that end-to-end behaviors be programmed. In CLB, applications data and control flow emerge as a natural consequence of content/context-driven execution of local behaviors.

7 Conclusion and Future Work

The growing ubiquity of sophisticated broadband sensor/actuator devices and the emergence of sensor-based pervasive environments are enabling new generations of applications based on seamless access, aggregation, and interactions. The defining characteristic of these applications is their ability to leverage the pervasive infrastructure to continuously manage, adapt, and optimize themselves to meet their objectives. Supporting these applications requires a programming and management paradigm where the behaviors as well as the interactions of applications elements (sensors, actuators, services, etc.) are dynamic and context, content and capability aware. Further, these interactions and the resulting flows are opportunistic.

In this paper we presented a programming model that enables such opportunistic flows in pervasive environments. The model is based on content-based discovery and routing services and provides two abstractions. *Associative rendezvous* provides an abstraction for content-based decoupled interactions. *Cascading local behaviors* build on associative rendezvous to enable opportunistic application flows where flows emerge as a result of context and content based reactive behaviors. A key characteristic of the pro-

posed model is that complex end-to-end applications flows do not have to be specified - only the local behaviors of sensor/actuator elements are independently defined and flows naturally emerge as a consequence of these local behaviors. This makes the model particularly suited to highly dynamic and ad hoc sensor/actuator-based environments. In this paper we also presented the design, prototype implementation and experimental evaluation of the Meteor programming framework and content-based middleware. We are currently working on a wider deployment of Meteor and the opportunistic flows model, and are exploring its extension to support coordinated flow with stronger guarantees.

References

[1] T. Abdelzaher, B. Blum, D. Evans, J. George, S. George, L. Gu, T. He, C. Huang, P. Nagaraddi, S. Son, P. Sorokin, J. Stankovic, and A. Wood. Envirotrack: Towards an environmental computing paradigm for distributed sensor networks. *IEEE ICDCS*, March 2004.

[2] B. Bayerdorffer. Distributed programming with associative broadcast. *Proceedings of the 27th Annual Hawaii International Conference on System Sciences, Volume 2: Software Technology (HICSS94-2), Wailea, HW, USA*, pages 353–362, 1994.

[3] P. Bhandarkar and M. Parashar. Semantic communication for distributed information coordination. *In Proceedings of the IEEE Conference on Information Technology, Syracuse, NY*, pages 149–152, September 1998.

[4] A. Carzaniga and A. L. Wolf. Content-based networking: A new communication infrastructure. *NSF Workshop on an Infrastructure for Mobile and Wireless Systems*, October 2001.

[5] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Fanklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. *In Proceedings of 2003 CIDR Conference*, 2003.

[6] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *TODS*, December 2003.

[7] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar. Next century challenges: Scalable coordination in sensor networks. *In Proceedings of MobiCOM '99, Seattle, Washington*, August 1999.

[8] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[9] G. Fox, S. Pallickara, and X. Rao. A Scaleable Event Infrastructure for Peer to Peer Grids. In *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pages 66–75, Seattle, Washington, USA, 2002. ACM Press.

[10] J. Gao and P. Steenkiste. Rendezvous points-based scalable content discovery with load balancing. *In Proceedings of the Fourth International Workshop on Networked Group Communication (NGC'02), Boston, MA*, pages 71–78, October 2002.

[11] D. Gay, P. D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. *In Proceedings of Programming Language Design and Implementation (PLDI) 2003*, June 2003.

[12] Gryphon: publish/subscribe over public networks. http://www.research.ibm.com/gryphon/papers/Gryphon-Overview.pdf.

[13] T. Iso, Y. Isoda, K. Otsuji, H. Suzuki, S. Kurakake, and T. Sugimur. Platform technology for ubiquitous services. *NTT DoComMo Tchnical Journal*, 5(1), June 2003.

[14] JavaSpaces. http://www.javaspaces.homestead.com/.

[15] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler. The emergence of networking abstractions and techniques in TinyOS, 2004.

[16] S. Madden, J. Hellerstein, and W. Hong. TinyDB: In-network query processing in tinyOS, version 0.4, September 2003.

[17] V. Matossian and M. Parashar. Autonomic optimization of an oil reservoir using decentralized services. *Proceedings of CLADE 2003, Seattle, WA, USA*, pages 2–9, June 2003.

[18] A. Omicini and E. Denti. From tuple spaces to tuple centers. *Science of Computer Programming*, 41:277 – 294, 2001.

[19] C. Schmidt and M. Parashar. Flexible information discovery in decentralized distributed systems. In *Proceedings of the 12th High Performance Distributed Computing (HPDC)*, pages 226–235. IEEE Press, June 2003.

[20] C. Schmidt and M. Parashar. Enabling flexible queries with guarantees in p2p systems. *Internet Computing Journal*, 8(3):19–26, 2004.

[21] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. *Proceedings of ACM SIGCOMM'02, Pittsburgh, PA*, pages 73–86, August 2002.

[22] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SICOMM'01 Conference*, pages 149–160, San Diego, California, August 2001.

[23] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, 1997.

[24] P. Wyckoff. T spaces. *IBM Systems Journal*, 37(3):454 – 478, 2001.

[25] Y. Yao and J. E. Gehrke. The cougar approach to in-network query processing in sensor networks. *Sigmod Record*, 31(3), September 2002.