Opportunistic Application Flows in Sensor-based Pervasive Environments

Nanyan Jiang, Cristina Schmidt, Vincent Matossian, and Manish Parashar

ICPS 2004
Outline

- Introduction to pervasive sensor-based applications
- Meteor: A content-based middleware for decoupled interactions
  - Content-based routing infrastructure
  - Programming model for decoupled interactions
  - Opportunistic application flows
  - Implementation and evaluation
Motivation

- Growing ubiquity of sensor/actuator devices with embedded communication and computation capabilities
- Emergence of pervasive applications
  - Sensors, actuators, services, resources – interact and collaborate to satisfy application goals
  - Examples:
    - Fire management applications
    - Emergency medical care
Sensor-based Pervasive Environments

- Ad hoc structures/behaviors
  - Interactions form and stop in an ad hoc manner
- Dynamic
  - Peers join and leave at any given time
- Unreliable
  - No delivery or routing guarantees
- Loosely-consistent
  - Global knowledge bound by scale of the system
- Heterogeneous
  - Rich variety of platforms, systems, and protocols
ORBIT: An Experimental Wireless Network Testbed

High-level Architecture of Proposed 2-Tier ORBIT Wireless Network Testbed
Enabling pervasive applications

- Pervasive applications
  - Ad hoc interactions between sensors and actuators
  - Bridge wired and wireless networks
  - Large, distributed, complex, heterogeneous, dynamic
  - Need for
    - Middleware architecture
      - Scalable and self-managing
      - Provide content-level addressing
      - Support asynchronous and decoupled interactions
    - Autonomic Programming model (self-* )
      - Defines interaction & coordination paradigm
        - Achieve global behavior without the need for global knowledge
Project AutoMate: Software Stack

NeTS Applications
(Autonomic Living, Ad hoc Control)

Coordinated Flows

Autonomic Elements, Emergent Flows/Opportunistic Interactions

Content Routing and Discovery, Associative Messaging

Self Organizing Content Overlay

Wireless/Wired Infrastructure

Ad Hoc Routing
Self Configuration

Secrecy, Authorization, Authentication, Trust

Programming Model

Orbit Testbed

Meteor Middleware Stack

Ontology, Taxonomy

The Applied Software Systems Laboratory
Meteor - Content-based Middleware

- Self-organizing overlay
  - Chord P2P overlay, each peer is a *rendezvous point*
- Content-based routing
  - Squid
- AR Messaging
  - Profile Manager
  - Matching engine
Self-organizing Overlay (Chord)

- A self-organizing P2P ring overlay
- Nodes and data have unique identifiers (keys), from a circular key space (0 to $2^m$)
- Each node maintains a routing table, called “finger table”
- A key is stored at the first node whose identifier is greater or equal than the key
- The request is routed to the neighbor node closer to the destination
- Routes in $O(\log n)$ hops

**Finger** = the successor of (this node id + $2^{i-1}$) mod $2^m$, $0 \leq i \leq m$

Routing from node 1 to node 6
Content-based Routing (Squid)

- Chord can route based on unique data identifiers only
- Squid uses a dimension-reducing indexing scheme
  - Can route based on keywords, partial keywords, wildcards and ranges
  - Uses the Space-Filling Curves for mapping keywords to identifiers
- Squid offers guarantees: all destinations matched by the keywords will be identified
Hilbert Space-Filling Curve (SFC)

- $f: \mathbb{N}^d \rightarrow \mathbb{N}$, recursive generation

Properties:
- Digital causality
- Locality preserving
- Clustering
Squid – Definitions

- **Keyword tuple (used to specify the destination)**
  - List of \(d\) keywords, wildcards and/or ranges
  - Example:
    - \((\text{temperature, celsius}), (\text{temp*}, *), (*, 10, 20-25)\)

- **Simple keyword tuple**
  - Contains only complete keywords

- **Complex keyword tuple**
  - Contains wildcards and/or ranges
Squid – Content-based Routing (1)

- Using simple keyword tuples
  - Routing to a single destination – equivalent to Chord lookup

Map the point (2, 1) to index 7 using the Hilbert Space Filling Curve (SFC).

Route to node 13 (the successor of the index 7)
**Using complex keyword tuples**

- Routing to multiple destinations – the straightforward solution

![Diagram showing complex keyword tuple and route to nodes storing clusters](image.png)

- Translate the query to relevant clusters on the SFC-based index space

- Route to the nodes that store the clusters

- Longitude

- Latitude

- Complex keyword tuple

- Destination nodes
Squid: Experimental Evaluation (1)

- Simulation
- Up to 5000 nodes, and up to $10^6$ keyword tuples
- Metrics:
  - Number of processing nodes
  - Number of data nodes
- Keyword tuples:
  - Keyword tuples with wildcards
  - Keyword tuples with ranges
- System size increases from 1000 to 5000 nodes, keys from $2*10^5$ to $10^6$
Squid: Experimental Evaluation (2)
Associative Rendezvous (AR)

- Content-based decoupled interactions:
  - All interactions are based on content, rather than names or addresses
  - The participants (e.g. senders and receivers) communicate through an intermediary, the *rendezvous point*
  - The communication is asynchronous. The participants can be decoupled both in space and time.

- Programmable reactive behaviors:
  - The reactive behaviors at the rendezvous points encapsulated within message => flexibility, expressivity, and multiple interaction semantics
Associative Rendezvous: Interaction Model

- **Messages:**
  - (header, action, data)
  - Symmetric post primitive

- **Associative selection**

- **Reactive behavior**

Profile = list of (attribute, value) pairs:

Example:

\<\text{(sensor\_type, temperature)}, \text{(latitude, 10)}, \text{(longitude, 20)}\>
Associative Rendezvous: An Illustrative Example

**AR: Associative Rendezvous**

- **post** \((p_1, p_2), \text{notify\_interest}(C1)\)
- **notify\_interest** \((C1)\)
- **post** \((p_1, *), \text{notify\_data}(C2)\)

C1 → C2
Associative Rendezvous: An Illustrative Example

AR: Associative Rendezvous

post \((<p_1, p_2>, \text{store}, \text{data})\)

\(<p_1, *> \rightarrow \text{notify\_data}(C2)\)

\(\text{notify}(C2)\)

C1

C2
Associative Rendezvous: An Illustrative Example

AR: Associative Rendezvous

\[
\langle p_1, p_2 \rangle \rightarrow \text{store} \rightarrow \text{data} \\
\text{post}(\langle p_1, * \rangle, \text{retrieve} (C2)) \\
\text{retrieve}(C2, \text{data})
\]
**Associative Rendezvous: An Illustrative Example**

**AR: Associative Rendezvous**

\[
\text{post}(<p_1, p_2>, \text{delete\_data (C1)}) \\
<p_1, p_2> \rightarrow \text{store} \rightarrow \text{data} \\
\text{post}(<p_1, *>, \text{delete\_interest (C2)}) \\
<p_1, *> \rightarrow \text{retrieve (C2)}
\]
Cascading Local Behaviors (CLB)

- An abstraction for content-based decoupled interaction with programmable reactive behaviors.

- A programming model, where the behaviors of individual application elements are locally defined in terms of local state, and context and content events, and result in data and interest messages being produced.

- Application flows emerge as a consequence of the cascading effect of local behaviors, without having to be explicitly programmed.
Cascading Local Behaviors – Example

Temperature Sensor

\[
\text{if} \ (\text{temp} > 80) \\
\text{then} \ \text{publish} \ \text{temp} 
\]

Thermostat

\[
\text{if} \ (\text{temp} > 85) \\
\text{then} \ \text{temp\_control} = \text{on}; \\
\text{publish} \ \text{temp\_control} 
\]

Window Actuator

\[
\text{if} \ (\text{temp\_control} == \text{on}) \\
\text{then} \ \text{close} \ \text{windows} 
\]

\[\text{meteor}\]

Temperature Sensor

\[\text{post} \ (\text{temp} = 91, \text{store})\]

Thermostat

\[\text{post} \ (\text{temp} > 85, \text{notify\_data})\]

\[\text{notify} \ (\text{thermostat})\]

\[\text{turn temp\_control} \ \text{on}\]

\[\text{post} \ (\text{temp\_control, store})\]

Meteor

\[\text{meteor}\]

\[\text{post} \ (\text{temp\_control, retrieve\_data})\]

Window Actuator

\[\text{retrieve} \ (\text{temp\_control})\]

\[\text{if} \ (\text{temp\_control} = \text{on}) \\
\text{then} \ \text{Close window}\]
Meteor: Implementation Overview

- Current implementation builds on JXTA
- Chord, Squid and AR Messaging layers are implemented as event-driven JXTA services
- Each layer exposes an API to the upper layer
- Chord uses
  - JXTA discovery mechanism to find other nodes already in the group
  - JXTA resolver service to send messages between peers
Meteor: Experimental Evaluation (1)

- Current Deployment
  - 64 node Linux cluster, 1.6 GHz Pentium IV, each node running at least one peer (rendezvous node)
  - 100 Mbps Ethernet interconnection network

- Experimental evaluation:
  - Chord lookup overhead as a function of system size
  - Content-based routing overhead
  - Associative Messaging overhead at each RP node:
    - querying the database – associative selection
    - constructing the notification messages – reactive behavior
Meteor: Experimental Evaluation (2)

Overlay network lookup overhead (Chord)

Content-based routing overhead (Squid)
Meteor: Experimental Evaluation (3)

Matching overhead at a single RP

Number of profiles stored locally

- 100 profiles
- 1000 profiles
- 10000 profiles
- 10000 profiles

Number of matches

Time (milliseconds)

- 0.1 0.5 1 5 10 20 30
- 0 200 400 600 800 1000 1200 1400 1600

Matching
overhead
at a
depend
Number of profiles stored locally
Conclusions

- Content-based, decoupled programming model is suited to address the challenges of pervasive applications
- A P2P infrastructure is a natural solution to implement the Associative Rendezvous abstraction
- JXTA is a convenient platform to develop decoupled, content-based middleware
More Information

- Meteor project page:
  - http://www.caip.rutgers.edu/TASSL/Projects/Meteor/

- Squid project page:
  - http://www.caip.rutgers.edu/~cristins/research.html

- Recent report:
  - http://www.jxta.org/universities/rutgers.html
Load Balancing (LB) – Why Necessary?

- The node identifiers (Chord) are uniformly distributed in the identifier space.
- The data to be stored in the system is not uniformly distributed in the $d$-dimensional space.
- The SFC preserves locality.

=>$\text{The SFC index is not uniformly distributed, and we need load balancing.}$
Load Balancing - Discussion

- Assume that all nodes have the same storage and computational capabilities
- The LB algorithms used balance the storage load, not the computational one
- Future directions:
  - Better LB algorithms, that take into consideration
    - The nodes’ heterogeneity
    - The “hot-spots” – some nodes will store more popular information, and they will have lots of requests
Load balancing – Algorithms

- **Load balancing at node join:**
  - generate more than one ID for the new node, send join requests in the network and join with the ID that places the node in the most crowded part of the network

- **Load balancing at runtime:**
  - run a local load balancing algorithm between neighbors (from time to time), and redistribute the load
  - use virtual nodes that can migrate to less loaded physical nodes
Load balancing – Simulation Results

The distribution of the keys in the index space. The index space was partitioned into 5000 intervals. The Y-axis represents the number of keys per interval.

The distribution of the keys when using only the load balancing at node join technique.

The distribution of the keys when using both the load balancing at node join technique, and the local load balancing.
Squid: Routing Optimization

- More than one cluster are typically stored at a node
- Not all clusters that are generated for a query exist in the network => optimize!
- SFC generation recursive => clusters generation is recursive => the process of cluster generation can be viewed as a tree
- Optimization: embed the tree into the overlay, and prune nodes during the construction phase
Squid: Routing Optimization - Example

Binary complex keyword tuple (011, *)

---

July 22nd 2004  Opportunistic flows in pervasive environments ICPS 2004

37/30
# Squid vs. Existing P2P Systems

<table>
<thead>
<tr>
<th>P2P System</th>
<th>Pros &amp; Cons</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unstructured</strong></td>
<td><strong>Pro:</strong> overlay easy to maintain, supports complex queries&lt;br&gt;<strong>Con:</strong> no search or cost guarantees</td>
<td>Gnutella – like systems</td>
</tr>
<tr>
<td><strong>Hybrid</strong></td>
<td><strong>Pro:</strong> supports complex queries&lt;br&gt;<strong>Con:</strong> not scalable</td>
<td>Napster, Morpheus</td>
</tr>
<tr>
<td><strong>Data-lookup</strong></td>
<td><strong>Pro:</strong> efficient lookup with guarantees&lt;br&gt;- complex queries not supported&lt;br&gt;- high structure overlay maintenance cost</td>
<td>Chord, CAN, Tapestry, Pastry</td>
</tr>
<tr>
<td><strong>Structured Keyword Search</strong></td>
<td><strong>Pro:</strong> supports more complex queries&lt;br&gt;(e.g. keywords, SFC-based systems support even partial keywords, wildcards and ranges)&lt;br&gt;<strong>Con:</strong> high structure overlay maintenance cost</td>
<td>Inverted Indices, PeerSearch, SFC-based systems (HP Labs, <strong>Squid</strong>)</td>
</tr>
</tbody>
</table>
Implementation Overview (cont’d)

- Squid uses Chord’s *lookup* primitive, to route clusters to their destination
- Squid’s API consists of one primitive:
  - `post (AR_Message)` – routes based on keywords extracted from the message’s profile
- AR Messaging Layer
  - API consists of a single function: `post (header, action, data)`
  - It currently uses a MySQL database as storage for profiles, and as a matching engine