Introduction

- Information dissemination solutions are needed by many distributed applications
  - Content delivery
    - News, alerts, stock market information, metadata, presence information,…
    - Monitor data (sensors)
    - Control data (actuators, robots,…)
- Motivates research and development of efficient distributed dissemination systems
  - Event-based systems and Publish/Subscribe
  - Reusable building blocks for high-level routing

Event-based Systems and Publish/subscribe

- Event delivery from publishers to subscribers
  - Event is a message with content
  - One-to-many, many-to-many
  - Builds on messaging systems and store-and-forward
- A frequently used communication paradigm
  - Decoupling in space and time
  - Solutions from local operation to wide-area networking
  - Proposed for mobile/pervasive computing
- The event service is a logically centralized service
  - Basic primitives: subscribe, unsubscribe, publish
- Various routing topologies and semantics
**Subscriptions**

- Subscriptions are described using filters
  - Filter: a stateless Boolean function
    - Defines a subspace of the content space
    - A single event is a point in the content space
  - Selects a subset of published events
  - Expressive interest definition and content-matching
  - Content model is typically typed tuples or XML

**Event Systems I**

- Traditional MoM systems are message queue based (one-to-one)
- Event systems and publish/subscribe are one-to-many or many-to-many
  - One object monitors another object
  - Reacts to changes in the object
  - Multiple objects can be notified about changes
- Events address problems with synchronous operation and polling
- In distributed environments a logically centralized service mediates events
  - anonymous communication
  - expressive semantics using filtering

**Event Systems II**

- Push versus Pull
- May be implemented using RPC, unicast, multicast, broadcast...
- Three main patterns
  - Observer design pattern
    - Used in Java / Jini
  - Notifier architectural pattern
    - Used by many research systems
  - Event channel
    - Used in CORBA Event/Notification Service
- Filtering improves scalability / accuracy
  - Research topic: content-based routing
**Tuple Spaces**

- Tuple-based model of coordination
- The shared tuple space is global and persistent
- Communication is
  - decoupled in space and time
  - implicit and content-based
- Primitives
  - In, atomically read and removes a tuple
  - Rd, non-destructive read
  - Out, produce a tuple
  - Eval, creates a process to evaluate tuples
- Examples: Linda, Lime, JavaSpaces, TSpaces

**Java Message Service (JMS)**

- Asynchronous messaging support for Java
- Point-to-point messaging
  - One-to-one
- Topic-based publish/subscribe
  - SQL for filtering messages at the topic event queue
  - One-to-many
- Message types:
  - Map, Object, Stream, Text, and Bytes
- Durable subscribers
  - Event stored at server if not deliverable
- Transactions with rollback

**OMG Distributed Data Service I**

- The Data Distribution Service for Real-Time Systems (DDS)
- The specification defines an API for data-centric publish/subscribe communication for distributed real-time systems.
- DDS is a middleware service that provides a global data space that is accessible to all interested applications.
- DDS uses the combination of a Topic object and a key to uniquely identify instances of data-objects.
- Content filtering and QoS negotiation are supported
- DDS is suitable for signal, data, and event propagation.
**Content-based Routing**

- Filters select events based on the content of event messages
- Can be seen as content-based addressing
- More expressive than topic, subject, or header-based routing
- Research projects and prototype systems
  - Siena, Rebeca, Hermes, ...
- Three central design considerations
  - Router topology
  - Interest propagation mechanism
  - Filtering language

**Example of Siena-style Content-based Routing**

- A filter is a stateless Boolean function that takes a notification as an argument
- Filter F1 is said to cover filter F2 if and only if all the notifications that are matched by F2 are also matched by F1.
- The covering relation is a partial order (transitive, anti-symmetric)
- If filters are organized in a graph based on the covering relation, several optimizations are possible.
- The **root set** of the graph is the **minimal cover set**
Data Structures for Cover based Routing

• Filters Poset
  – For hierarchical and peer-to-peer routing (acyc/cyc)
  – A directed acyclic graph structure that stores the direct predecessors and successors for each filter.
  – Each filter has a **subscribers** set.
  – Two first levels are used to compute forwarding information (**forwards** sets)
    • Filter S from X
      – All filters from X that are covered by S are removed.
      – Never forward S from X to X or to interfaces where a covering filter has been already sent.
    • Unsubscription may result in uncovered filters to be sent (direct successors of deleted filter)
  • Poset-derived forest
    – Data structure with support for fast add/del operations.
    – Forest representation based on covering relation
    – Local clients, hierarchical, and peer-to-peer routing

Routing Blocks

This is a set of generic building blocks for filter cover-based routing.
Can be extended with optimizations such as pruning and caching.

Filter Merging

• Filter merging is useful, because it allows to further remove redundancy and keep the number of elements minimal.
• There are many ways to perform filter merging.
• We assume that a **merge**($F_1, F_2$) procedure exists
  – Returns a single merged filter $F_M$.
  – $F_M$ covers both $F_1$ and $F_2$.
• A merge of two or more filters is called a **merger**.
• A merger is either **perfect** or **imperfect**.
  – A perfect merger does not result in false positives or negatives, whereas an imperfect merger may result in false positives.
Dynamic Filter Merging Example

Filter Merging II

• Requirements for filter merging
  – Merging must be transparent for applications and routers.
  – An insert of \( x \) may result in a new merged node \( M = \text{merge}(x, y) \), but after the delete of \( x \) the resulting node must cover \( y \).

• Formal framework with filter merging rules
  – Keep track of the components of a merger and the nodes that are covered by the merger
    • Example: \( x \) and \( y \) are components of \( M \).
  – Does not specify how the selection of merging candidates is done or how possible re-merging after delete is realized

• Filter merging may be applied in different places in the event router.

Rules for Merging I

\( F_1 \) \( \equiv \) \( F_2 \) \( \Rightarrow \) Del(\( F_2 \)) on data structure, if they are from the same interface.

\( F_1 \) \( \equiv \) \( M_1 \) \( \Rightarrow \) Del(\( M_1 \))

\( \text{Del}(M_1) \) \( \Rightarrow \) Remove \( M_1 \) and reset its covered nodes and components (first rule is applied as well)
Rules for Merging II

- **M1** ⊇ **F1** => Add **F1** to **M1**'s covered set

- **M1** ⊇ **M2** => Add the components and covered nodes of **M2** to **M1** and remove **M2**.

- Del(x) and x is a component of a merger **M1** => Remove **M1** from MR, reset **M1**'s components and covered nodes. This makes them available for re-merging.

Dynamic Merging

- Goal to have a simple and efficient mechanism
  - Optimal merging is a hard problem
  - Should be linear time for practical usage
  - Opportunistic operation
- Keep track of merged elements, elements covered by mergers, and non-covered elements
- Perform merging only on the root set (or 2 first levels)
  - Triggered when the root set changes due to addition or removal of an element
  - Add: given new element, scan root set for merger, then scan for covered elements
  - Del: simply remove element from sets
    - Restore uncovered elements (and non-deleted components of merger) to the root set and attempt to merge them

Merging Blocks

- Aggregate Merger
  - Filters Poset
    - k neighbours + local clients
  - Root Merger
  - Poset-derived Forest
    - n local clients

- Non-redundant Forest
  - 1 master router
    - k slave routers

- Root Merger
- Slave

Filter Language and Merging

- The formal framework supports different filter merging mechanisms.
  - Hardcoded rules, merging procedures, rules from ontologies
    - Merging rules may also be approximate (resulting in false positives)
- We use a simple typed tuple model for experiments.
  - Each filter is a set of attribute filters with unique name and type.
  - Relational operators are supported.
    - Selected from \{<, >, ≤, ≥, =, ≠, [a, b]\} using a uniform distribution.
    - Two filters are mergeable if they have only one differing attribute filter that can be merged.
Root Merger Benchmarks

1. Add or Add/Remove scenario
2. Matching test

Workload generator → Filters → Forest → Root Merger → Correctness testing

2D Filters

Add scenario time
Add scenario root set size
Forest Merged forest

Add/Remove scenario total time
Add/Remove scenario total ops
Forest Merged forest

100 filters replaced in Add/Remove

PosetBrowser

Articles on Routing Blocks

Chaining Forests

DoubleForest: Chaining Two Forests

- DoubleForest data structure
  - Combines two poset-derived forests for generic context matching.
  - One forest for queries, one for profiles.
- Support for both subspace matching and temporal profiles.
- Mappings are updated during add/del.
- Optimizations possible using forest structure.
- Experimental results indicate that overlap based matching has more overhead than cover based matching.

Overview

Queries

- \( Q_1 = [0,10] \)
- \( Q_2 = [12,20] \)
- \( Q_3 = [2,7] \)
- \( Q_4 = [15,22] \)
- \( Q_5 = [16,20] \)

Profiles

- \( P_1 = [1,5] \)
- \( P_2 = [5,10] \)
- \( P_3 = [5,25] \)
- \( P_4 = [17,20] \)
- \( P_5 = [7,9] \)

Profile forest (P)

- \( P_3 \rightarrow P_1 \)
- \( P_4 \rightarrow P_2 \)
- \( P_3 \rightarrow P_4 \)
- \( P_5 \rightarrow P_1 \)

Query forest (Q)

- \( Q_1 \rightarrow Q_2 \)
- \( Q_3 \rightarrow Q_5 \)
- \( Q_4 \rightarrow Q_1 \)
- \( Q_3 \rightarrow Q_4 \)
- \( Q_5 \rightarrow Q_1 \)

Mappings

- \( M_{QP} : P \rightarrow \mathcal{P}(Q) \)
- \( M_{QP} : Q \rightarrow \mathcal{P}(P) \)

- \( M_{QP}(Q_1) \rightarrow \{P_1, P_2, P_5\} \)
- \( M_{QP}(Q_2) \rightarrow \{P_4\} \)
- \( M_{QP}(Q_3) \rightarrow \{\emptyset\} \)
- \( M_{QP}(Q_4) \rightarrow \{P_4\} \)
- \( M_{QP}(Q_5) \rightarrow \{P_4\} \)

- \( M_{QP}(P_1) \rightarrow \{Q_1\} \)
- \( M_{QP}(P_2) \rightarrow \{Q_1\} \)
- \( M_{QP}(P_3) \rightarrow \{\emptyset\} \)
- \( M_{QP}(P_4) \rightarrow \{Q_2, Q_4, Q_5\} \)
- \( M_{QP}(P_5) \rightarrow \{Q_1\} \)

Chained Forests

- Generalizes to multiple sets, each corresponding to a forest in a chain of forests
- Separates mappings from the forests.
- Improved insertion and lookup cost. Improved or degraded deletion cost and structure size depending on the workload. Sparse bitstrings may be used to alleviate space concerns.
- Nature of optimizations suggest that copes well with self-similar workload.
- Future work investigates how to compact mappings
### Results

![Graph](chart.png)

**Insertion and deletion (6000 elements)**

<table>
<thead>
<tr>
<th></th>
<th>1AF</th>
<th>2AF</th>
<th>3varAF</th>
<th>3AF</th>
</tr>
</thead>
<tbody>
<tr>
<td>DF</td>
<td>1769497</td>
<td>14023</td>
<td>3484</td>
<td>87026</td>
</tr>
<tr>
<td>Poset</td>
<td>544265</td>
<td>243826</td>
<td>5998</td>
<td>208447</td>
</tr>
<tr>
<td>Forest</td>
<td>779112</td>
<td>185538</td>
<td>5852</td>
<td>6000</td>
</tr>
</tbody>
</table>

Size after inserting 6000 elements

<table>
<thead>
<tr>
<th></th>
<th>DF</th>
<th>Poset</th>
<th>Forest</th>
</tr>
</thead>
<tbody>
<tr>
<td>1AF</td>
<td>655</td>
<td>5724</td>
<td>317695</td>
</tr>
<tr>
<td>2AF</td>
<td>675</td>
<td>3066</td>
<td>637056</td>
</tr>
<tr>
<td>3varAF</td>
<td>667</td>
<td>3200</td>
<td>406348</td>
</tr>
<tr>
<td>3AF</td>
<td>719</td>
<td>673</td>
<td>937292</td>
</tr>
</tbody>
</table>

Time (ms) for 30 000 lookups for 3000 elements

### Filters and Context

- We propose to represent context using filters
  - Support for points and subspaces (for example ranges)
  - Use filters for both queries and profiles.
    - A query defines a collection and is matched against profiles.
    - A profile describes the context of an object
    - Two semantics: cover and overlap
- A set of user interest $\rightarrow$ context query $\rightarrow$ filter $\rightarrow$ subspace of context space
- A set of context metadata $\rightarrow$ context profile $\rightarrow$ filter $\rightarrow$ subspace of context space

### DataSpace Architecture

- **Client Terminal**
  - Object and dir. sync.
  - When to sync?
- **Collection**
  - Contents based on context
- **Synchronization Server**
  - Add new object with context profile to the directory
  - Keep track of context subscriptions and notify when an object is added/removed that matches with the subscr.
- **Profile Store**
  - Policies ACRs
  - Profile & Query Store

### Articles

**Publish/Subscribe Spam Prevention**

**Pub/Sub Spam**
- Typical email spam is unsolicited bulk mail that clutters inboxes
- SIP spam is envisaged to take many forms
  - IM spam, presence spam, voice spam
- Pub/sub spam has many dimensions
  - Both control and content aspect
  - Different characteristics than email/SIP
    - Different targeting mechanism
      - Filters define end-points and delivery paths instead of explicit addresses
    - Multi-hop networks
    - Many-to-many nature
  - If filters are used for clustering/ranking/recommendations, we can have "filter engine" spam

**Lessons from Email Spam**
- Spammers have vast resources at their disposal
  - Zombie machines, cheap labor,..
  - No technique alone seems to be sufficient
  - Obfuscatory techniques such as open relays, spoofing, and zombies make it difficult to track spammers
- Sender authentication is key
  - Well-known solutions include IP-based and crypto-based techniques: SPF, Yahoo’s DomainKeys

**Solution Space**
- Solution space includes the following
  - Distributed Blackhole Lists, which collect IP addresses of known spammers
  - Crypto-based sender verification
  - Greylisting, in which messages with unseen (IP, sender, receiver) triplets are delayed
  - Rate limiting
  - Proof-of-work techniques
  - Content filtering for outbound and inbound messages
  - Access control using buddy-lists and social networks
    - These can also be used for spamming
Content-based Spam

- The key issue in pub/sub spam is event replication
- A powerful technique for scalable dissemination, but it is also a possible spamming technique
- To ensure wide-scale dissemination of spam, spammers must
  - Estimate filters that are widely subscribed
  - Create content that matches those filters
- In order to find popular filters, spammers may infiltrate the broker network
  - Bogus brokers monitor traffic, possibly drop messages
  - Redundancy is needed to detect and mitigate bogus brokers

Preventive Measures

- The aim of infrastructure-based spam prevention is to incorporate preventive measures into the core routing network
  - Drop unwanted messages as near the source as possible
- Current systems have not explicitly addressed spam prevention
  - Some solutions available in the form of security services.
  - Real workloads and traces needed
- Key aims:
  - Preventing bogus brokers and publishers
  - Preventing black hole queries and advertisements

Identity-based Spam Prevention

Identity

Verifies and/or requires

Lookup service

Lookup & reporting

B1

B2

B3

P

Principles for Spam Prevention

- Publisher and broker authentication to prevent bogus brokers and publishers
- Distributed blacklisting using DNS or a DHT
- Filter set generalization using techniques such as filter covering and merging
- Detecting spam sources that try to systematically cover the content space by flooding
- Preventing black hole advertisements from brokers and black hole subscriptions from clients
- Keeping popular filters secret
Mobility Support

Challenges with Mobility Support

• How to cope with mobile users?
  – Disconnected operation
    • Buffering and queue management
  – Mobile subscribers / publishers
    • Handover protocol for relocating subscriptions and updating the topology
    • Multiple indirection points on the overlay network
    • Covering/merging complicate mobility support
  – General requirements
    • fast convergence of the subscription topology
    • mobility-safety: no false negatives

Example Handover

Articles

Towards Pub/Sub Internetworking

Rethinking Naming, Trust, and Primitives

- Current Internet architecture is sender-oriented
  - Unwanted traffic
  - Connectivity problems
- We propose a future Internet that gives more control to the receiver
  - Publish/Subscribe model
  - Only end-points described using interests and local links
  - Basic routing using flat self-certifying labels
  - Data-centric routing, forwarding, rendezvous
- Two approaches:
  - Incremental with overlay networks
  - Radical clean-slate approach with a new networking stack: Internet without IP
- Related work: UIP, FARA, TRIAD, Nimrod, i3, DOA, ROFL, DONA, AIP

Publish / Subscribe

1. Establish (hash, ID)
2. Route subscribe and establish forwarding path from Pub to Sub
3. Publish (hash, ID)
4. Verify ID in publications

Looking at Layers

- Link layer
- Network layer (IP)
- Transport layer (TCP/IP)
- Application layer

- Link layer
- Network layer (IP)
- Pub/sub Transport layer
- Application layer

- Overlay pub/sub
- Transport layer
- Pub/sub layer
- Application layer
Summary

• Publish/subscribe supports many-to-many information networking
• Construction of routing systems using reusable building blocks, namely posets and forests
• Recent research on
  – Spam prevention
  – Pushing pub/sub down the stack
  – Efficient matching
• Live demos on the web
  – http://www.tml.tkk.fi/~starkoma/fc/