Middleware-Konzepte

Publish/Subscribe-Systeme

Dr. Gero Mühl

Kommunikations- und Betriebssysteme
Fakultät für Elektrotechnik und Informatik
Technische Universität Berlin
Agenda

> Introduction
> Notification Selection
> Data and Filter Models
> Content-Based Routing Algorithms
> Efficient Use of Multicast
> Matching Algorithms
> Structuring Publish/Subscribe Systems
> Formal Specification of Publish/Subscribe Systems
Introduction
Motivation

> Trend towards information-driven applications
  > Automated (or require at least reduced user interaction)
  > Depend on external data sources
  > Selection, filtering, and combination of data
  > React to changes

> Information no longer static but rather flowing from producers to consumers!
  > Mechanisms to access static data (queries) different from those controlling the flow of information (subscriptions/filters)!
Motivation (contd.)

> Application examples
  > Personalized news services
  > Stock monitoring software
  > Distributed auction platforms
  > Monitoring and adaptation of distributed systems
  > Wireless sensor networks (data fusion etc.)
  > Zero latency enterprise (ZLE)
Client/Server Systems

> Two Roles
  > **Clients request** data or functionality from a server
  > **Servers reply** to clients’ requests
  ⇒ Request/Reply interaction

> Characteristics
  > Synchronous communication (initiated by the client)
  > Direct Addressing
  ⇒ Tight coupling

1. Request:
   Current quote of stock „DT“?

2. Reply:
   12,52€ at 15:02:34
Client/Server Systems (contd.)

> Well suited for accessing static information

> Disadvantages regarding information-driven applications
  > Data source must be queried periodically (polling)
  > Communication occurs regardless whether new data is available
  > Every query costs (network bandwidth, computing power)
  > Every client queries individually
  > Low latency implies high polling frequency
  > Inaccuracy chains along multiple information flow steps

⇒ Trade-off between accuracy and scalability
⇒ Client/server systems do not suit information-driven applications
Publish/Subscribe Systems

> Two Roles
  > **Producers** publish notifications
  > **Consumers** subscribe to notifications

> Notification service (NS) delivers notifications

> Characteristics of Pub/Sub
  > Asynchronous communication (initiated by the producer)
  > Indirect Addressing (anonymous)
  \[\Rightarrow\] Loose coupling

- Consumer subscribed to “Stock = DT ∧ Price > 13€”
- Consumer subscribed to “Stock = DT”
Publish/Subscribe Systems (contd.)

> Advantages regarding information-driven applications
  > Communication only occurs when new data is available
  > Low latency because initiation done by producer
  > Distribution to many consumers enables optimization
    (e.g., exploitation of multicast)

⇒ **Publish/Subscribe better suits information-driven applications!**
Publish/Subscribe API

interface 
Subscriber

+process(event:Event):void

eventService

0..*

subscribers

eventService

interface 
EventService

+publish(event:Event):void
+subscribe(subscription:Subscription):void
+unsubscribe(subscription:Subscription):void

interface 
Publisher
Pub/Sub Industry Products

> TIBCO Rendezvous
> CORBA Notification Service implementations (e.g., by Iona)
> Java message service (JMS) implementations (e.g., SUN ONE Message Queue)
> Oracle Advanced Queuing
> IBM WebSphere MQ (formerly MQSeries)
> MS .NET notifications
> ...
Pub/Sub Research Prototypes

> SIENA  (Univ. of Colorado at Boulder)
> Jedi   (Politecnico di Milano, Italy)
> Elvin  (DSTC, Australia)
> REBECA (TU Darmstadt and TU Berlin, Germany)
> Gryphon (IBM TJ Watson Research Center, USA)
> X²TS  (TU Darmstadt, Germany)
> Hermes (Univ. of Cambridge, UK)
> …
Application Scenarios

> Securities data feed handler publish the latest stock prices to hundreds of traders on a trading floor simultaneously
> Materials movement systems distribute data to various materials handlers, controllers, and tracking systems on a factory floor
> Inventory levels flow continuously to accounting, purchasing, marketing, and other departments in a retail store
> Network components are automatically monitored for optimizing performance and detecting failures
> …
Enterprise Application Integration (EAI)

> Integration of new and legacy applications
> Heterogeneous environment
  > Different protocols
    > Java RMI, Sockets, CORBA IIOP, SMTP, HTTP etc.
  > Different message formats
    > XML, EDI, proprietary format etc.
> Different operating systems and hardware platforms
> …

> Grown applications result in “Application Spaghetti”
  > At most $n^2/2$ interfaces and network connections
  ⇒ Point-to-point connections quickly become infeasible!
Application Spaghetti
EAI Architectures

> EAI main application and origin of message-oriented middleware in general and publish/subscribe in particular
> Reason is decoupling of senders and receivers

> Two main architectures
  > Hub-and-Spoke
  > Message Bus
Hub-and-Spoke

> Centralized architecture
> Reduces number of network connections and interfaces
> Decoupling of sender and receiver
> Most functionality embedded into the server
  > Authentication and authorization
  > Guaranteed message delivery
  > Message transformation and routing
Message Busses

- **Decentralized** architecture
- Messaging functionality embedded into all nodes
- Often rely on multicast communication
- More complete decoupling of senders and receivers
- No single point of failure or bottleneck
Zero Latency Enterprise

“Zero latency is the real-time, enterprise wide dissemination of new information distributed in such a way that allows businesses to react quickly to it, driving the competitive business advantage to its ultimate limits” (Paul Larson, Talarian Corporation)

“Instantaneous awareness and appropriate response to events across an entire enterprise” (Roy Schulte Vice President, World Services, Gartner)

According to Gartner, a real-time enterprise is event-driven [3].
Information Dissemination

> Information is disseminated to large set of consumers
> Optimal technique depends on
  > Number of producers
  > Number of consumers
  > Types of information
  > Interests of consumers
  > …

⇒ Data Delivery Options
Data Delivery Options

pull
- aperiodic
  - 1:1
  - Request/Reply
- periodic
  - 1:1
  - Polling

push
- aperiodic
  - 1:1
  - Polling with snooping
- periodic
  - 1:n
  - Callback, Email

Proposed by Franklin and Zdonik [1]
Is it Push or Pull++?

> Web browser periodically requests web page ⇒ pull
> But this looks to the user like push!
> Low latency implies high polling rate
> Architecture used by *Pointcast*
Web Caching

> Data is **pushed** from the web server to the caches

> Client is redirected to nearest cache

> E.g., text is retrieved from web server but images from nearest cache (→ Akamai)
Stock Monitoring / Automated Stock Trading

> Application requires low latency
> E.g., stocks are bought or sold
  > if prices rise above or fall under a given price
  > if prices rise or fall by at least a given amount in some time interval (3% in 15 minutes)
> Arbitrage trading exploits pricing differences at different exchanges
> Price movement of different stocks can be put into relation
> Publish/subscribe prevalent in the financial industry!
  > E.g., TIBCO Rendezvous used for the NASDAQ trading system
Ubiquitous Computing

> Raises new requirements
> Limited bandwidth, connectivity, processing power, and battery capacity
  ⇒ Information filtering becomes more important
  ⇒ Data aggregation on the server-side
> **Context-aware services**
  > Location, speed etc.
  ⇒ Spatial subscriptions

(photomontage)
Related Work

- Active databases
- GUI programming
- Event-triggered real-time systems
- Monitoring and debugging
- Workflow systems
- …
Active Databases

> Allow to define **triggers**
  (aka. **event condition action rules**)
> Triggers can be executed if data changes (i.e., on update, insert, or delete of table rows) or if a transaction starts, commits, or aborts
> Active functionality today incorporated in most databases

> Trigger Syntax in Oracle
  > `{BEFORE|AFTER} {INSERT|DELETE|UPDATE} ON <table_name> [REFERENCING [NEW AS <new_row_name>] [OLD AS <old_row_name>]]
  [FOR EACH ROW [WHEN (<trigger_condition>)]]} <trigger_body>`
Graphical User Interfaces (GUIs): Listener Pattern

> **Register/notify** used for a long time (e.g., Java AWT)
> GUI components (e.g., buttons) emit events
> Events (e.g., button pressed) correspond to state changes of the respective component
> Event *listeners* register event handlers *directly at the event source* (i.e., the button)
> Handlers are called if matching event is raised
Event-Driven Real-Time Systems

- **Real-time systems** must obey timing conditions
  - E.g., earliest start time and deadlines for tasks

- **Time-triggered** real-time system
  - Predefined **static** schedule (task execution plan)
  - E.g., sensor is read every 50ms

- **Event-driven** real-time system
  - No predefined schedule
  - Tasks are executed when events occur
  - E.g., sensor issues interrupt if temperature changes
  - More difficult to analyze due to dynamic behavior
Bibliography


   http://www.gartner.com/reprints/tibco/110277.html
Notification Selection
Notification Selection

> Selecting the interesting notifications

> Expressiveness of notification selection is crucial for a notification service!

> Trade-Off: **Expressiveness vs. Scalability**

> Common approaches

  > Channels

> Subjects

> Event Types

> Content-based Filtering

Increasing expressiveness
Channels

- Channels categorize notifications (weather, news, stock quotes etc.)
- Producer publishes notification with respect to a channel
- Simple, little expressive model
- Producers determines channel
  - Producers and consumers not fully decoupled
- Easily mappable to IP-multicast
- Used by the CORBA Event Service
Subjects (also known as Topics)

> **Subjects** categorize notifications hierarchically
  > Used by many commercial systems (e.g., TIBCO Rendezvous)

> **Wildcards**
  > *News.Stock Market.Quotes.* (select all child subjects)
  > *News.Politics.* (select whole subtree)
Subjects (contd.)

- Germany
- Politics
- US
- Telekom
- Siemens
- Quotes
- Stock Market
- News
Subjects (contd.)

> More expressive than channels
> Subject determined by producer
  ⇒ Still no full decoupling between producers and consumers

> Problem: How to divide the notification space into subjects?
  > Division has implications on the possible selections!
    > News.Stock Market.Quotes.Siemens or
  > Subjects only suited for categorizing along a single dimension
  > Changes to the subject tree have implications on consumers!
Subjects (contd.)

> Mapping to multicast more complicated
  > Implementations heavily depend on wildcard semantics

> Potential multicast-based implementations
  > For every subject and every subtree a multicast group
    > Notification published multiple times
    > Consumer subscribes to one group
  > Multicast group for every subject
    > Notification published only once
    > Consumer subscribes to several groups
      (e.g., to all of a subtree)
  > Delivery to all nodes and apply local filtering
  > ...
Type-Based Filtering

> Allows for **type-based categorization** of notifications
> Usually common abstract superclass *Event*
> *Event* is subclassed to generate specific classes of events (e.g., *StockQuote*)
> Producers publish event objects
> Consumers subscribe to classes of events
  ⇒ Subclasses are also delivered
    (this offers some limited kind of extensibility)
Type-Based Filtering (contd.)

- **Single inheritance** is similar to subjects
  - Several possibilities to structure the class hierarchy
- **Multi inheritance** might be a solution but has its own drawbacks

![Event hierarchy diagram]

**VS.**

- HardwareEvent
- SoftwareEvent
- PerformanceEvent
- FaultEvent
Content-Based Filtering

- Based on a **data model** with corresponding **filter model**
  - E.g., name/value pairs and conjunctive attribute filters
- Filters evaluate the whole content of notifications using logical expressions comprised of filter predicates
  - E.g., \(\{(\text{Stock} = \text{Telekom} \land \text{Price} > 15€)\}\)
- Hence, a filter \(F\) is a function that maps notifications to the Boolean values *true* and *false*
- Set of matching notifications \(N(F) = \{n \mid F(n) = \text{true}\}\)
Content-Based Filtering (contd.)

> Used by, e.g., CORBA Notification Service and Java Message Service (JMS)

> Most expressive but complex
  > Cannot be easily mapped to IP-Multicast
    (#groups may grow exponentially in #filters)
  > Centralized implementation not scalable to wide-area scenarios

⇒ Powerful distributed infrastructure needed
## Notification Selection Summary

<table>
<thead>
<tr>
<th></th>
<th>Channels</th>
<th>Subjects</th>
<th>Types</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expressiveness</td>
<td>Low</td>
<td>Moderate</td>
<td>Moderate</td>
<td>High</td>
</tr>
<tr>
<td>Mapping to Multicast</td>
<td>Easy</td>
<td>Moderate</td>
<td>Moderate</td>
<td>Complex</td>
</tr>
<tr>
<td>Decoupling / Flexibility</td>
<td>Low</td>
<td>Moderate</td>
<td>Moderate</td>
<td>High</td>
</tr>
<tr>
<td>Administration</td>
<td>Easy</td>
<td>Moderate</td>
<td>Moderate</td>
<td>Easy</td>
</tr>
<tr>
<td>Complexity of Infrastructure</td>
<td>Low</td>
<td>Moderate</td>
<td>Moderate</td>
<td>High</td>
</tr>
</tbody>
</table>
Bibliography


Data and Filter Models
Data and Filter Models

> Determine the expressiveness of content-based notification selection

> Examples
  > Tuples/Templates
  > Name-value pairs/Conjunctive Attribute Filters
  > XML/XPath
  > Event Objects/Filter Objects
Tuples

> Notifications are **tuples** (ordered sequences of attributes)
>   (“Stock Quote”, ”Telekom”, “15€”)

> **Templates** used for matching
>   (“Stock Quote”, “Telekom”, *)
>   Similar to a *query-by-example mask*

> Template matches notification iff
>   1. Template has same number of attributes as notification
>   2. Every attribute in the template matches the attribute at the same position in the notification
>   > Most common operators: equality, prefix, and don’t care (*)
Tuples (contd.)

- Known from LINDA Tuplespaces (JavaSpaces, TSpaces, etc.)
- Used by JEDI (Java Event-based Distributed Infrastructure)
- Simple but not very expressive model
- Model does not allow for optional attributes
Name/Value Pairs

> Notifications consist of set of **attributes**
> Attributes are **name/value pairs**

```{(Type, Quote), (Name, “Telekom”), (Price, 23.24)}
↑   ↑
Name  Value```
Name/Value Pairs (contd.)

> Filters are conjunctions of attribute filters
  > \( F = f_1 \land \ldots \land f_n \)
  > Filter is satisfied if and only if all attribute filters are satisfied

> Attribute filter applies a constraint to a named value
  > E.g., \((\text{Price} > 20\text{€})\)
  > Set of satisfying values \(n(f_i)\)
  > Usually at most one attribute filter per attribute allowed

> E.g., \{\((\text{Type} = \text{Quote}) \land (\text{Name} = \text{“Telekom”}) \land (\text{Price} > 20\text{€})\)\} is satisfied by
  \{\((\text{Type}, \text{Quote}), (\text{Name}, \text{“Telekom”}), (\text{Price}, 23.24)\)\}

> Model used by many pub/sub systems, e.g., SIENA
XML/XPath

> Notifications are modeled as XML documents

```xml
<notification>
  <auction endtime="05/18/02 22:17:42" minprice="$50">
    <seller name="Smith" id="1234"/>
    <item>
      <cpu manufacturer="AMD" type="Athlon"
           socket="Socket A" clock="800"/>
    </item>
  </auction>
</notification>
```

> Filters are modeled as XPath queries

```xml
  e.g., /notification[auction/item/cpu[@manufacturer="AMD"]]
```
XML/XPath (contd.)

> XPath expressions are very expressive
> Can also be used to select only parts of notifications
> Expressiveness must be limited to make routing optimizations feasible
Objects

> Notifications and filters are modeled as objects (e.g., as JAVA classes)
> Common superclasses
  > Event Baseclass of all events
  > Filter Baseclass of all filters
  > Subscription and Advertisement are subclasses of Filter
> Good integration with object-oriented programming languages
Implementation of *StockQuote* Class

> StockQuote subclasses Event

```java
public class StockQuote implements Event {
    String symbol;
    double price;
    long time;
    public String getSymbol() {
        return symbol;
    }
    ... 
}
```
Implementation of *EventClassFilter*

> *EventClassFilter* subclasses *Filter*

```java
public class EventClassFilter() implements Filter {
    Class eventClass;

    public EventClassFilter (Class eventClass) {
        this.eventClass = eventClass;
    }

    public boolean match(Event e) {
        return eventClass.isInstance(e);
    }
}
```

```java
eventClassFilter filter =
    new EventClassFilter(StockQuote.class);
```
Filter and Event Implementation (contd.)

```java
public class StockQuoteFilter extends EventClassFilter {
    public StockQuoteFilter() {
        super(StockQuote.class);
    }

    boolean matchStockQuote(StockQuote sq) {
        return true;
    }

    public boolean match(Event e) {
        if (super.match(e))
            return matchStockQuote((StockQuote)e);
        else
            return false;
    }
}
```
Filter and Event Implementation (contd.)

```java
public static void main(String[] args) {
    ...
    StockQuoteFilter filter = new StockQuoteFilter() {
        boolean matchStockQuote(StockQuote sq) {
            return sq.getSymbol().equals("DT") &&
                    sq.getPrice() > 20;
        }
    }
    ...
};
```
Objects (contd.)

> Necessary **serialization** and **deserialization** of events and filters might lead to degraded performance!
> Routers and consumers must have access to class implementations
> Routing optimizations difficult if filter objects contain arbitrary program code
> Solution: Tests are implemented by the objects

```java
public boolean identical(Filter f) {
    if (f instanceof StockQuoteFilter)
        return ((StockQuoteFilter)f).getSymbol().equals(this.symbol);
    else
        return false;
}
```
Bibliography


Content-Based Routing
The Need for a Distributed Implementation

> Disadvantages of a *centralized* notification service
  > Single point of failure
  > Bottleneck wrt. bandwidth and computing power
  > No exploitation of locality although most interactions will be local
  \[ \Rightarrow \] Neither reliable nor scalable to wide-area scenarios

> Service must be distributed and replicated
  \[ \Rightarrow \] Multiple cooperating service instances necessary!
Broker Network

> Service instances called **brokers**
> Each broker manages a set of exclusive **local clients** to which it forwards matching notifications it processes
> Notifications are stepwise **forwarded** through broker network

> **Cyclic forwarding** and **delivery of duplicates** must be avoided!
  > Easy in *acyclic* topologies but these provide no fault tolerance
  > Complex in topologies with *cycles*
Flooding

> Each notification is delivered to all brokers
  > A broker forwards an incoming notification to all neighbors if it was received from a local client and to all other neighbors if it was received from a neighbor

> Advantages
  > No routing tables for neighbor brokers necessary
  > Subscriptions become active immediately
  > Implementations can easily exploit IP-Multicast

> Disadvantage
  > Many notifications may be processed and forwarded unnecessarily wasting computing power and network bandwidth
Efficiency of Flooding

> Efficient if
  > most notifications need to be delivered to a majority of brokers or
  > subscriptions are highly transient

> Not efficient if
  > filters are highly selective or
  > locality among subscriptions exists
Content-Based Routing

> Each broker manages a filter-based routing table
> A routing table $T_B$ consists of a set of routing entries
> A routing entry is a pair $(F, D)$ consisting of a filter $F$ and a destination $D$ which can be a local client or a neighbor broker
  ⇒ *Local* (not shown in Fig.) and remote routing entries
Content-Based Routing (contd.)

1. $n \in N(F)$
2. $n \notin N(G)$
3. $n \notin N(H)$

Broker $B_1$

Forwarding Engine

Routing Table

$(F, B_3)$
$(G, B_4)$
$(H, B_3)$
Notification Forwarding in Acyclic Topologies

> Each broker forwards an incoming notification to all neighbor brokers and local clients for which there is a matching routing entry in the routing table.
>
> If the notification was received from a neighbor broker, it is not forwarded to that neighbor.
Routing Algorithms

> Routing tables must be updated if subscriptions are issued or revoked
  > *Update processes* triggered
  > Many update processes might be in progress currently
  ⇒ *Routing algorithm*
⇒ Subscriptions become active *gradually*
  ⇒ *Race conditions* and *latency*
  ⇒ Fully active after corresponding update process terminated

> Forwarding of unsubscriptions only done for efficiency
  > Delivery of non-matching notifications can be prevented locally
Routing Algorithms (contd.)

- A *correct* routing algorithm ensures that
  1. Each update process terminates within finite time
  2. After an update process terminated, all published notifications matching this subscription are delivered to the subscribing client
  3. No duplicates and no non-matching notifications are delivered
Routing Algorithms (contd.)

> **Trade-Off**

> Flooding notifications vs.
filtering at intermediate brokers and updating of routing tables

> Optimal approach heavily depends on current setting!
⇒ In dynamic environments adaptive solutions necessary
Simple Routing

> Only assumption: Filters can be uniquely identified
> Each subscription stored in every routing table
  > Size of routing tables grows linearly in the number of active subscriptions and the number of brokers
> Each routing table affected by a new/cancelled subscription
  > \#brokers – 1 update messages necessary if update information is not batched
⇒ Not scalable!

1. \textit{sub}(F)
2. \textit{sub}(G)
Simple Routing (contd.)

> New subscription flooded into the broker network
  > If a broker receives a subscription $F$ from a neighbor $B$,
    > it adds an entry $(F, B)$ to its routing table and
    > forwards the subscription $F$ to all its other neighbors

> Unsubscription flooded into the broker network
  > If a broker receives an unsubscription $F$ from a neighbor $B$,
    > it removes the entry $(F, B)$ from its routing table and
    > forwards the unsubscription $F$ to all its other neighbors
Advanced Content-Based Routing

> Goals
  > Smaller routing tables
  > Reduced filter forwarding overhead

> Possible Solutions
  > Routing algorithms that exploit similarities and merge subscriptions
  > Integration of advertisements (announcements of producers)
  > Trade accuracy vs. efficiency
    > perfect routing: Notifications are forwarded only if a matching subscription exists in respective subnet
    > imperfect routing: Notifications may be forwarded without any matching subscription
Identity-Based Routing

> Uses *identity tests* for routing decisions

\[ F \equiv G \iff N(F) = N(G) \]

> Avoids

  > Forwarding of identical filters
  > Routing entries with identical filters for the same neighbor broker
Covering-Based Routing

> Uses *covering tests* for routing decisions

\[ F \text{ covers } G \iff N(F) \supseteq N(G) \]

> Avoids

  > Forwarding of covered filters
  > Routing entries with covered filters for the same neighbor broker

> Forwarding of subscriptions similar to identity-based routing

> Forwarding of unsubscriptions more complex due to uncovered subscriptions
Covering-based Routing (contd.)

> Processing of a new subscription $F$ from a neighbor $B$
> Routing entries regarding $B$ whose filter is covered by $F$ are dropped
> $F$ is forwarded only to those other neighbors to which no covering subscription was forwarded before which is still active
Covering-Based Routing (contd.)

> Processing of an unsubscription $F$ from a neighbor $B$
  > $F$ is forwarded only to those other neighbors to which no covering subscription was forwarded before which is still active
  > Along with $F$, uncovered subscriptions must be forwarded
Merging-Based Routing

> Filters can be merged
  > perfectly \( N(F) = N(G) \cup N(H) \)
  > imperfectly \( N(F) \supset N(G) \cup N(H) \)

> Merging generates new covers
Merging-Based Routing (contd.)

> Merging-based routing is *perfect* iff
  > only perfect mergers are generated *and*
  > mergers are forwarded and cancelled in a way such that no unnecessary notifications are received later on
> … otherwise it is *imperfect*
Perfect Merging-Based Routing

> Possible implementation on top of covering-based routing
  > Brokers merge suitable routing entries \((F_1, D), \ldots, (F_n, D)\) with the same destination \(D\) to a single entry \((F, D)\) such that \(F\) is a perfect merger of \(\{F_1, \ldots, F_n\}\)
  > Merged entries are removed from routing table
  > Merger is added to routing table and forwarded like a normal subscription received from \(D\)
Hierarchical Routing Algorithms

> One broker is labeled as root broker $R$
> Any notification and filter is forwarded “upwards” to $R$
> Notifications are only forwarded “downwards” if a matching filter exists
> Leads to smaller routing tables but increases the load imposed on brokers on higher levels such as the root broker

1. $\text{sub}(F)$
2. $\text{pub}(n)$
   $n \in \mathcal{N}(F)$
Hybrid Routing Algorithms

> Combine hierarchical with peer-to-peer routing
Advertisements

> Producers specify the notifications they will publish
> Advertisements are filters that are forwarded like subscriptions by the help of a second routing table
> Subscriptions are only forwarded to those destinations for which there is an overlapping advertisement

> Test for overlapping: $N(F) \cap N(G) \neq \emptyset$
Advertisements (contd.)

Advertisements Routing Table

(B3, F)

(1, B1, F)

(2, F)

(G, B4)

(H, B5)

Advertisements

Routing Table

Subscription

Advertisements

G

H
Advertisements (contd.)

> New advertisement
  > If $B$ receives a new advertisement from $B'$, $B$ forwards those overlapping subscriptions to $B'$ which can newly be served

> Cancelled advertisement
  > If $B$ receives a cancelled advertisement from $B'$, $B$ drops those overlapping subscriptions which can no longer be served
Advertisements (contd.)

> Race condition
  > Published notifications may be dropped if they match only a newly servable subscription that has not yet arrived!

> Potential solutions
  > Keep notifications that match only the new advertisement until all newly servable subscriptions have arrived → corrupts FIFO producer ordering
  > Block a producer that publishes a notification that matches only the new advertisement until all newly servable subscriptions have arrived → introduces some synchronicity
Topology Changes

> Easy approach preserving an acyclic topology
  > A new broker $B$ is connected as a leaf
    (i.e., to exactly one broker $B'$)
    > $B'$ sends all interesting subscriptions to $B$
      > Simple Routing: all subscriptions
      > Identity-based Routing: identical subscriptions are suppressed
      > Covering-based Routing: covered subscriptions are suppressed
    > If advertisements are used, they are exchanged instead of subscriptions
  > Only leaf brokers can be removed
Supporting Routing Optimizations

> Complexity of routing optimizations depends on the underlying data/filter model
  > In the most general case not computable
  > E.g., covering test among relational expressions is \( \text{NP}\)-complete
> Scalability vs. Expressiveness

⇒ Expressiveness must be limited to a sensible level

> We assume conjunctive filters consisting of attribute filters with at most one attribute filter per attribute in the following
Identity among Filters

> Two filters $F_1$ and $F_2$ are *identical*, written $F_1 \equiv F_2$, iff
> (a) they contain the same number of attribute filters and
> (b) for each attribute filter in $F_1$ there is an attribute filter in $F_2$ such that these attribute filters are identical

> Example of identical filters

$$F_1 = \{x \geq 2\} \land \{y > 5\}$$

$$\equiv \equiv \equiv$$

$$F_2 = \{x \geq 2\} \land \{y > 5\}$$
Overlapping among Filters

> Two filters $F_1$ and $F_2$ are disjoint iff for an attribute filter in $F_1$ there exists an attribute filter in $F_2$ that constrains the same attribute such that these attribute filters are disjoint.

> Two filters are overlapping iff they are not disjoint.

> Example for disjoint filters

\[ F_1 = \{x \geq 2\} \land \{y > 5\} \]
\[ F_2 = \{x < 1\} \land \{y = 7\} \]
Covering among Filters

> A filter $F_1$ \textit{covers} a filter $F_2$, written $F_1 \sqsupseteq F_2$, iff for each attribute filter in $F_1$ there is a covered attribute filter in $F_2$

> Example of covering filters

\[ F_1 = \{x \geq 2\} \land \{y > 5\} \]
\[ \sqsupseteq \sqsupseteq \sqsupseteq \]
\[ F_2 = \{x = 4\} \land \{y = 7\} \land \{z \in [3,5]\} \]
Filter Lattice

> Filters are arranged in a lattice according to covering relation

> Lattice is updated according to new and cancelled subscriptions

> Lattice is used by covering-based routing algorithm

\[
F_1 := \{x > 5\} \\
F_2 := \{x > 8\} \\
F_3 := \{x = 9\} \\
F_4 := \{x = 6\} \\
F_5 := \{x > 6 \land y > 7\} \\
F_6 := \{x > 7 \land y = 9\} \\
F_7 := \{x = 9 \land y = 9\}
\]
Merging of Filters

> Two filters $F_1$ and $F_2$ can be merged *perfectly* iff they are identical in all but a single attribute filter.

> Merging Example

$F_1 = \{x \geq 2\} \land \{y < 4\}$

$F_2 = \{x \geq 2\} \land \{y > 6\}$

$F_3 = \{x \geq 2\} \land \{y \not\in [4,6]\}$ (Merger)
Example: Geographic Information Systems (GIS)

> \( F = \{(\text{Type} = \text{TrafficInformation}) \land (\text{Location around}(\text{Frankfurt},50\text{km}))\}\)

> \( G = \{(\text{Type} = \text{TrafficJam}) \land (\text{Length} \geq 5\text{km?}) \land (\text{Location around}(\text{Darmstadt},20\text{km}))\}\)

\( \Rightarrow \) \( F \) covers \( G \)

> \( H = \{(\text{Type} = \text{TrafficJam}) \land (\text{Location around}(\text{Frankfurt},40\text{km}))\}\)

> \( I = \{(\text{Type} = \text{TrafficJam}) \land (\text{Location around}(\text{Wiesbaden},40\text{km}))\}\)

\( \Rightarrow \) \( H \) and \( I \) can be merged imperfectly
Bibliography


Efficient Use of Multicast
Use of Multicast Mechanisms

- Seems naturally to use IP-multicast for publish/subscribe
  but
  - IP-Multicast not reliable
  - Number of groups limited
    (in theory $2^{24}$, in practice much smaller)
  - Tunnels necessary to connect not multicast-enabled areas

- Easy solution for channels
  - Use for each channel a separate multicast group
Use of Multicast Mechanisms (contd.)

> Solution for subjects more complex
  > Depends on the possible filtering expressions (e.g., wildcards)
  > E.g., one multicast group for each subject and subtree

> Content-based filtering most complex
  > The only simple solution is flooding
Ideal Multicast

> Direct send a notification to all matching brokers
> May require $2^N$ groups ($N$ number of all brokers)
  > Number of all subsets of all brokers
> Not practical for large numbers of brokers

> Alternatives
  > Reduce group precision (i.e., use overly broad groups)
  > Do multiple sends (i.e., split set of destinations)
  > Send using multiple hops (i.e., send to neighbors only)
Clustered Group Multicast (CGM)

- Divides brokers into $C$ mutually exclusive subsets (clusters) with $K = N / C$ brokers
- Requires $C$ sends and $n = C \cdot 2^K = C \cdot 2^{N/C}$ groups
- Reduces number of groups by a factor of $2^{C/C}$
- Reduces efficiency of multicast
- Still not feasible for larger $N$ (e.g., $N = 100$, $C = 10 \Rightarrow n = 10240$ groups necessary)
- Clustering assignment important
  - Subscriptions (clients’ interests)
  - Geographic location of brokers
Threshold Clustered Group Multicast (TCGM)

> Sends to whole cluster if number of matching brokers in the respective cluster exceeds a certain threshold $T$
> Requires only groups for all subsets of the cluster with cardinality less than $T$, i.e., $C \cdot \sum_{i=1}^{T} \binom{K}{i}$ groups
Neighbor Matching Algorithm

> Each broker only sends to matching neighbors
> Requires multiple hops
> Multicast group for all combinations of neighbors
> Advantages
  > More scalable for larger $N$
    (only $N \cdot 2^k$ groups necessary, $k$ number of neighbors)
  > Knowledge about groups can be local
> Disadvantages
  > Requires extra bandwidth on links between brokers and network
  > Matching must be executed at every broker introducing additional delays
## Comparison of Multicast Strategies

<table>
<thead>
<tr>
<th></th>
<th>Ideal</th>
<th>Flooding</th>
<th>CGM</th>
<th>TCGM</th>
<th>Neighbor</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Precision</strong></td>
<td>1</td>
<td>$P_N$</td>
<td>1</td>
<td>$[(T+1)/K,1]$</td>
<td>1</td>
</tr>
<tr>
<td><strong>Multicasts</strong></td>
<td>1</td>
<td>1</td>
<td>1…C</td>
<td>1…C</td>
<td>1 per hop</td>
</tr>
<tr>
<td><strong>Groups</strong></td>
<td>$2^N$</td>
<td>1</td>
<td>$C \cdot 2^K$</td>
<td>$C \cdot \sum_{i=1}^{T} \binom{K}{i}$</td>
<td>$N \cdot 2^k$</td>
</tr>
<tr>
<td><strong>Groups/Broker</strong></td>
<td>$2^{N-1}$</td>
<td>1</td>
<td>$2^{K-1}$</td>
<td>$\sum_{i=1}^{T-1} \binom{K-1}{i}$</td>
<td>$k \cdot 2^{K-1}$</td>
</tr>
<tr>
<td><strong>Manageability</strong></td>
<td>Hard++</td>
<td>Trivial</td>
<td>Moderate</td>
<td>Moderate</td>
<td>Moderate</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$N$</th>
<th>Total number of brokers</th>
<th>$K = N / C$</th>
<th>Number of brokers in a cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k$</td>
<td>Number of neighbors</td>
<td>$1 \leq T \leq K$</td>
<td>Threshold</td>
</tr>
<tr>
<td>$C&gt;1$</td>
<td>Number of clusters</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Bibliography


Matching Algorithms
Matching Problem / Forwarding Problem

- **Matching Problem**
  - How can one efficiently determine all matching subscriptions given a notification?

- **Forwarding Problem**
  - How can one efficiently determine the set of destinations (neighbor brokers and clients) which have a matching subscription given a notification?
  - Note that it is not necessary to find all matching subscriptions!
Matching Algorithms

> Many known algorithms
  > Brute Force
  > Predicate counting
  > Decision trees
  > Decision diagrams
  > …
Brute Force

> Matches a notification against all subscriptions separately
> Most simple implementation
> No complicated data structures necessary
> Relations among predicates not exploited
  > E.g., the same predicate may be evaluated multiple times
> Can be applied to any kind of filter
  > E.g., not restricted to conjunctive filters
Predicate Counting Algorithm

> A conjunctive filter is matched iff all its predicates are satisfied
> The algorithm counts the number of predicates of a filter that are satisfied by a given notification
> Requires for each predicate a list of filters it occurs in
> Avoids testing a predicate multiple times
> Evaluates every predicate at most once
> Works best for conjunctive filters
Predicate Counting Algorithm (contd.)

<for each filter a counter is initialized to zero>
FOR EACH <predicate matched by given notification> DO
   <increment the counter of all filters containing this predicate>
END
<all filters whose counter is equal to their number of predicates are matched by the given notification>
Predicate Counting Algorithm (contd.)

> If attribute filters are used, a *multi-level index structure* with specialized indexes can be used to efficiently determine the satisfied predicates

> **Attribute name index** used to find all predicates constraining a given attribute

> **Operator index** used to find all predicates with a certain operator

> **Value index**

  > Allows to find all satisfied predicates

  > Depends on the characteristics of the operator (e.g., hashing for equality tests)
Predicate Counting (contd.)

$F_1 := \{\text{Stock} = \text{“DT”} \land \text{Price} > 12\}$

$F_2 := \{\text{Stock} = \text{“MSFT”} \land \text{Price} > 14\}$

$F_3 := \{\text{Stock} = \text{“TIBX”} \land \text{Price} < 18\}$
Decision Trees

> Predicates and subscriptions are arranged in a graph
>   > Non-leaf nodes are tests
>   > Leaf nodes are subscriptions
>   > Edges are test results
> Graph is traversed in depth-first order
> Reached subscriptions are matched by notification

Tested notification

(Stock, “DT”)  
(Price, 14)  
(Volume, 2,000,000)

$$
F_1 := \{ \text{Stock} = \text{“DT”} \land \text{Price} < 15 \} \\
F_2 := \{ \text{Stock} = \text{“DT”} \land \text{Price} < 12 \} \\
F_3 := \{ \text{Stock} = \text{“TIBX”} \land \text{Price} > 16 \} \\
F_4 := \{ \text{Volume} > 1,000,000 \}
$$
Decision Trees (contd.)

> Efficient for equality tests because
  > test results are exclusive and
  > max. 2 matching outgoing edges exist
> Commonalities of subscriptions are not adequately exploited (subtree repetition)
> Limited expressiveness
> Restricted to conjunctive filters
> Expensive updates
> Used by Gryphon [1]
Binary Decision Trees

\[
F_1 := \{\text{Stock} = "DT" \land \text{Price} < 12\}
\]

\[
F_2 := \{\text{Stock} = "DT" \land \text{Price} \geq 12\}
\]

\[
F_3 := \{\text{Stock} = "TIBX" \land \text{Price} > 16\}
\]

\[
F_4 := \{\text{Volume} > 1,000,000\}
\]
Binary Decision Diagrams

> BDDs are directed acyclic graphs (DAGs)
> Each *non-terminal* node corresponds to a filtering predicate and has two outgoing edges
  > Low edge (predicate not satisfied)
  > High edge (predicate satisfied)
> 2 *terminal nodes* represent constants *true* and *false*
> Can be constructed from binary decision tree

\[ F := \{ Stock=\text{"TIBX"} \land ( Price > 15\$ \lor Volume > 1\text{Mio.\$}) \} \]
Corresponding If-then-else Program

1. If (Stock="TIBX") then 2 else 4;
2. If (Price>15) then 5 else 3;
3. If (Volume>1Mio.$) then 5 else 4;
4. Output 0;
5. Output 1;
Evaluation of Binary Decision Diagrams

Stock = "TIBX"
Price > 15$
Volume > 1Mio.$

match

Stock, "TIBX"
Price, 16
Volume, 500,000
Evaluation of Binary Decision Diagrams

Stock = "TIBX"
Price > 15$
Volume > 1Mio.$

match

Stock, "TIBX"
Price, 14
Volume, 2,000,000
Evaluation of Binary Decision Diagrams

Stock, "TIBX"
Price, 14
Volume, 500,000

no match
Binary Decision Diagrams (contd.)

> Ordered BDDs (OBDDs)
  > Nodes are numbered from 1 to \( n \) such that for every path the number assigned to the nodes is monotonically increasing.
  > For every filter the node with the lowest number is called the *output node*; it represents the outcome of the filter.

> Reduced BDDs (RBDDs)
  > No redundant nodes
  > No isomorphic subgraphs

> Size (#nodes) of BDDs may be exponential in the number of predicates
  > Ordering has great effect on size
  > Finding optimal ordering is NP-hard
Binary Decision Diagrams (contd.)

> Ordered BDDs can be evaluated bottom up

```
FOR v := n DOWNTO 1 DO
    IF v is terminal node
        THEN value[v] := label[v];
    ELSE BEGIN
        a := eval[v]; // evaluate filter predicate
        value[v] := a AND value[high[v]] OR
                    NOT a AND value[low[v]];
    END
    FI
END
```
Binary Decision Diagrams (contd.)

\[ F_1 := \{ \text{Stock} = "TIBX" \land \text{Price} > 15 \lor \text{Stock} \neq "TIBX" \land \text{Volume} > 1,000,000 \} \]

\[ F_2 := \{ \text{Stock} = "TE" \land \text{Price} > 15 \} \]

Order: \( \text{Stock} = "TIBX", \text{Stock} = "TE", \text{Price} > 15, \text{Volume} > 1,000,000, 1, 0 \)

\( \text{ITE}(a, h, l) = (\neg a \land l) \lor (a \land h) \)

Stock, "TIBX"
Price, 16
Volume, 500,000

If-then-else

output nodes
Bibliography


