Content-Based Distributed Event-Based System

Master CILS — Module IAAIO, Revision : 2169

Denis Conan

October 2018
Most of the content of these slides is extracted from the following references:

Outline

1. Content-based data and filter models
2. Distributed Event-Based Systems
3. Formal specification of distributed routing
4. Distributed routing algorithms
5. Lab on content-based filtering with the muDEBS framework
1 Content-based data and filter models

1.1 Data model and Filter model
1.2 Structured records
1.3 Semi-structured records
1.4 Objects
1.1 Data model and Filter model

- **Data model:** how the content of notifications is structured

- **Filter model:** how subscriptions can be specified
  - How notifications can be selected by applying filters that evaluate predicates over the content of notifications
1.1.1 Tuples

- **Data model:**
  - A notification is a tuple: an ordered set of attributes

- **Filter model:**
  - A subscription is defined as a template
  - The attributes of notifications and templates are matched to each other according to their position

- **Example:** the notification \((\text{StockQuote}, \text{"Foo Inc"}, 45)\) is matched by the subscription template \((\text{StockQuote}, \text{"Foo Inc"}, \*)\)
  - Tuples with templates provide a simple model that is not flexible
    - Because attributes cannot be optional
1.2 Structured records

1.2.1 Data model
1.2.2 Filter model
1.2.3 Identity, overlapping, covering of attribute filters
1.2.4 Routing optimisations with identity
1.2.5 Routing optimisations with covering
1.2.6 Covering with types and comparison
1.2.7 Covering with intervals and strings
1.2.8 Covering with sets
1.2.9 Routing optimisations with overlapping
1.2.10 Routing optimisations with merging
1.2.1 Data model

- A notification $n$ is a nonempty set of attributes $\{a_1, ..., a_n\}$
- $a_i$ is a (name,value) pair: $(n_i, v_i)$
- Attribute names are unique: $i \neq j \Rightarrow n_i \neq n_j$
- Example of notification: $\{(\text{type, StockQuote}), (\text{name,"Infineon"}), (\text{price, 45.0})\}$
- More powerful than tuples since attributes can be optional in subscriptions and notifications
1.2.2 Filter model

- **Filter** $F = \text{boolean function applied to a notification } n: F(n) \rightarrow \{true, false\}$

- **Attribute filter**: triple $A_i = (n_i, Op_i, C_i)$
  with $n_i = \text{attribute name}$, $Op_i = \text{test operator}$, $C_i = \text{value for the test}$

- $L_A(A_i) = \text{set of values } v_i \text{ that cause an attribute filter to match attribute } n_i$
  - $L_A(A_i) = \{v_i | Op_i(v_i, C_i) = true\}$
  - Usually $L_A(A_i) \neq \emptyset$

- **Compound filter** = conjunction of simple filters: $F = A_1 \land ... \land A_n$
  - *E.g.*, $(\text{type} = \text{StockQuote}) \land (\text{name} = \text{"Foo Inc"}) \land (\text{price} \notin [30, 40])$

- The set of matching notifications $N(F) = \{n | F(n) = true\} \subseteq N$

+ Attributes can be optional in the notification
+ New attributes can be added without affecting existing filters
1.2.3 Identity, overlapping, covering of attribute filters

- **Identity:**
  - $A_1 \equiv A_2$ iff $n_1 = n_2 \land L_A(A_1) = L_A(A_2)$
  - *E.g.*, $(\text{price} \in \{20, 21, 22, 23, 24, 25\})$ is identical to $(\text{price} \in [20, 25])$

- **Overlapping:**
  - $A_1 \cap A_2$ iff $n_1 = n_2 \land L_A(A_1) \cap L_A(A_2) \neq \emptyset$
  - *E.g.*, $(\text{price} > 25)$ overlaps $(\text{price} \in [20, 30])$

- **Covering:**
  - $A_1 \sqsupseteq A_2$ iff $n_1 = n_2 \land L_A(A_1) \supseteq L_A(A_2)$
  - *E.g.*, $A_1 = (\text{price} > 10)$ covers $A_2 = (\text{price} \in [20, 30])$

- **Disjoint:**
  - $A_1 \nmid A_2$ iff $n_1 = n_2 \land L_A(A_1) \cap L_A(A_2) = \emptyset$
  - ${\text{price} < 10}$ and ${\text{price} > 20}$ are disjoint
An identity test among filters is necessary to implement identity-based routing to avoid redundant routing entries and unnecessary forwarding of (un)subscriptions.

Given two filters $F_1 = A_1^1 \land ... \land A_n^1$ and $F_2 = A_1^2 \land ... \land A_m^2$ that are conjunctions of attribute filters with at most one attribute filter per attribute,

$F_1 \equiv F_2$ iff

they contain the same number of attributes filters $\land (\forall i, \exists j : A_i^1 \equiv A_j^2)$

E.g., $F_1 = \{x = 4\} \land \{y > 5\}$ not identical to $F_2 = \{x = 4\} \land \{y > 5\} \land \{z \in [3, 5]\}$
1.2.5 Routing optimisations with covering

- A covering test among filters is necessary to implement covering-based routing to avoid redundant routing entries and unnecessary forwarding of (un)subscriptions ∧ to get rid of the obsolete\(^1\) routing entries.

- Given two filters \(F_1 = A^1_1 \land ... \land A^1_n\) and \(F_2 = A^2_1 \land ... \land A^2_m\) that are conjunctions of attribute filters with at most one attribute filter per attribute,

\[F_1 \supseteq F_2 \text{ iff } \forall i, \exists j : A^1_i \supseteq A^2_j\]

- E.g., \(F_1 = \{x = 4\} \land \{y > 5\}\) covers \(F_2 = \{x = 4\} \land \{y > 5\} \land \{z \in [3, 5]\}\)

- E.g., \(F_3 = \{x \geq 2\} \land \{y > 5\}\) covers \(F_4 = \{x = 4\} \land \{y = 7\} \land \{z \in [3, 5]\}\)

---

1. A routing entry covers another routing entry, which becomes obsolete.
1.2.6 Covering with types and comparison

- $n_1 = n_2$

- Covering among notification types
  - A notification $n$ is an instance of Type $T$: $n$ instanceof $T$
  
<table>
<thead>
<tr>
<th>$A_1$</th>
<th>$A_2$</th>
<th>$A_1 \sqsupseteq A_2$ iff</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$ instanceof $T_1$</td>
<td>$n$ instanceof $T_2$</td>
<td>$T_1 = T_2 \lor T_1$ supertypeof $T_2$</td>
</tr>
</tbody>
</table>

- Covering among comparison constraints on simple values
  
<table>
<thead>
<tr>
<th>$A_1$</th>
<th>$A_2$</th>
<th>$A_1 \sqsupseteq A_2$ iff</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \neq c_1$</td>
<td>$x &lt; c_2$</td>
<td>$c_1 \geq c_2$</td>
</tr>
<tr>
<td>$x &gt; c_1$</td>
<td>$x &gt; c_2$</td>
<td>$c_1 \leq c_2$</td>
</tr>
</tbody>
</table>

- E.g., $A_1 = (x \neq 15)$ and $A_2 = (x < 10) \implies A_1 \sqsupseteq A_2$
- E.g., $A_1 = (x > 10)$ and $A_2 = (x > 20) \implies A_1 \sqsupseteq A_2$
1.2.7 Covering with intervals and strings

- \( n_1 = n_2 \)

- **Covering among interval constraints on simple values**

\[
\begin{array}{c|c|c}
A_1 & A_2 & A_1 \sqsupseteq A_2 \text{ iff} \\
\hline
x \in l_1 & x \in l_2 & l_1 \supseteq l_2 \\
\hline
x \notin l_1 & x \notin l_2 & l_1 \subseteq l_2 \\
\end{array}
\]

- E.g. \( A_1 = (x \in [3, 10]) \) and \( A_2 = (x \in [4, 6]) \) \( \implies A_1 \sqsupseteq A_2 \)

- **Covering among constraints on strings**

\[
\begin{array}{c|c|c}
A_1 & A_2 & A_1 \sqsupseteq A_2 \text{ iff} \\
\hline
s \text{ hasPrefix } S_1 & s \text{ hasPrefix } S_2 & S_2 \text{ hasPrefix } S_1 \\
\hline
s \text{ hasPostfix } S_1 & s \text{ hasPostfix } S_2 & S_2 \text{ hasPostfix } S_1 \\
\hline
s \text{ hasSubstring } S_1 & s \text{ hasSubstring } S_2 & S_2 \text{ hasSubstring } S_1 \\
\end{array}
\]

- E.g. \( A_1 = (s \text{ hasPrefix } "abc") \) and \( A_2 = (s \text{ hasPrefix } "abcd") \) \( \implies A_1 \sqsupseteq A_2 \)
1.2.8 Covering with sets

- $n_1 = n_2$

- Covering among set constraints on simple values

<table>
<thead>
<tr>
<th>$A_1$</th>
<th>$A_2$</th>
<th>$A_1 \sqsupseteq A_2$ iff</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \in M_1$</td>
<td>$x \in M_2$</td>
<td>$M_1 \supseteq M_2$</td>
</tr>
<tr>
<td>$x \notin M_1$</td>
<td>$x \notin M_2$</td>
<td>$M_1 \subseteq M_2$</td>
</tr>
</tbody>
</table>

- Covering among set constraints on multi values

<table>
<thead>
<tr>
<th>$A_1$</th>
<th>$A_2$</th>
<th>$A_1 \sqsupseteq A_2$ iff</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X \subset M_1$</td>
<td>$X \subset M_2$</td>
<td>$M_1 \supseteq M_2$</td>
</tr>
<tr>
<td>$X \text{ contains } a_1$</td>
<td>$X \supseteq M_2$</td>
<td>$a_1 \in M_2$</td>
</tr>
<tr>
<td>$X \supseteq M_1$</td>
<td>$X \supseteq M_2$</td>
<td>$M_1 \subset M_2$</td>
</tr>
<tr>
<td>$X \text{ notContains } a_1$</td>
<td>$X \text{ disjunct } M_2$</td>
<td>$a_1 \in M_2$</td>
</tr>
<tr>
<td>$X \text{ disjunct } M_1$</td>
<td>$X \text{ disjunct } M_2$</td>
<td>$M_1 \subset M_2$</td>
</tr>
<tr>
<td>$X \text{ overlaps } M_1$</td>
<td>$X \text{ overlaps } M_2$</td>
<td>$M_1 \supseteq M_2$</td>
</tr>
</tbody>
</table>
1.2.9 Routing optimisations with overlapping

- An overlapping test among filters is necessary to use advertisements in subscription-based routing optimisations.

- Advertisement and subscription routing tables are used to route (un)subscriptions from consumers to producers.
  - A subscription can be served by an advertisement if both overlap.

- Given two filters $F_1 = A_1^1 \land ... \land A_n^1$ and $F_2 = A_1^2 \land ... \land A_m^2$ that are conjunctions of attribute filters with at most one attribute filter per attribute,
  - $F_1$ and $F_2$ are disjoint iff $\exists i, j : (n_i^1 = n_j^2) \land (L_A(A_i^1) \cap L_A(A_j^2) = \emptyset)$
    - E.g., $F_1 = \{x \geq 2\} \land \{y > 5\}$ and $F_2 = \{x < 1\} \land \{y < 7\}$ are disjoint because $\{x \geq 2\}$ and $\{x < 1\}$ are disjoint.
  - $F_1$ and $F_2$ overlap iff $\nexists i, j : (n_i^1 = n_j^2) \land (L_A(A_i^1) \cap L_A(A_j^2) = \emptyset)$
    - E.g., $F_1 = \{x \geq 2\} \land \{y > 5\}$ and $F_2 = \{x < 5\} \land \{y < 7\}$ because $\{x \geq 2\}$ overlaps $\{x < 5\}$ and $\{y > 5\}$ overlaps $\{y < 7\}$.
### 1.2.10 Routing optimisations with merging

#### Merging of conjunctive filters

- A merging test among filters is necessary to implement merging-based routing to reduce the number of subscriptions and advertisements stored by brokers.

- **Examples:**
  - \( F_1 = \{x = 5\} \land \{y \in \{2, 3\}\} \) and \( F_2 = \{x = 5\} \land \{y \in \{4, 5\}\} \) can be merged to \( F = \{x = 5\} \land \{y \in \{2, 3, 4, 5\}\} \)
  - \( F_1 = \{y = 3\} \land \{x = 5\} \) and \( F_2 = \{y = 3\} \land \{x \neq 5\} \) can be merged to \( F = \{y = 3\} \)

- **Example of perfect merging rules for attribute filters**

<table>
<thead>
<tr>
<th>( A_1 )</th>
<th>( A_2 )</th>
<th>Condition</th>
<th>( A_1 \cup A_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x \in M_1 )</td>
<td>( x \in M_2 )</td>
<td>-</td>
<td>( x \in M_1 \cup M_2 )</td>
</tr>
<tr>
<td>( x \notin M_1 )</td>
<td>( x \notin M_2 )</td>
<td>( M_1 \cap M_2 = \emptyset ) ( M_1 \cap M_2 \neq \emptyset )</td>
<td>( \exists x ) (i.e., no att. filter) ( x \notin M_1 \cap M_2 )</td>
</tr>
<tr>
<td>( X \text{ overlaps } M_1 )</td>
<td>( X \text{ overlaps } M_2 )</td>
<td>-</td>
<td>( X \text{ overlaps } M_1 \cup M_2 )</td>
</tr>
<tr>
<td>( X \text{ disjunct } M_1 )</td>
<td>( X \text{ disjunct } M_2 )</td>
<td>( M_1 \cap M_2 = \emptyset )</td>
<td>( \exists X ) (i.e., no att. filter)</td>
</tr>
</tbody>
</table>
1.3 Semi-structured records

1.3.1 Data model
1.3.2 Filter model
1.3.1 Data model

- Notification = XML document = set of elements arranged in a tree
  - Element = set of attributes + subordinate child elements
    - Attributes = pairs \( (name, value) \)
    - Sibling attributes can have same name \( \Rightarrow \) names address sets of attr.

```xml
<notification>
  <auction endtime="05/18/02 22:17:42"
    minprice="50">
    <seller name="Smith" id="1234"/>
    <item>
      <board ...
    </item>
    <item>
      <cpu manufacturer="AMD"
        type="Athlon" clock="800"/>
    </item>
  </auction>
</notification>
```
1.3.2 Filter model I

- A filter model uses a path expression (e.g., XPath)
- A filter is a conjunction of path filters: \( F = \bigwedge_i P_i \)
- A path filter \( P = (S, C) \) consists of an element selector \( S \) and an element filter \( C \)
  - An element selector selects a subset of the elements of a notification
    - An absolute path: e.g. `/notification/auction/item/cpu`
    - An abbreviated path: e.g. `//cpu`
  - An element filter is a conjunction of a nonempty set of attribute filters:
    \( C = \bigwedge_i A_i \)
    - e.g. `[@manufacturer = “AMD” \& @clock ≥ 700]`
- Example of path filter:
  `/notification/auction/item/cpu[@manufacturer = “AMD” \& @clock ≥ 700]`
1.3.2 Filter model II

- $L_A(A)$: set of all values that cause an attribute filter $A$ to match an attribute

- $A_1 = (n_1, Q_1)$ covers $A_2 = (n_2, Q_2)$, $A_1 \sqsupseteq A_2$ iff $n_1 = n_2 \land L_A(A_1) \supseteq L_A(A_2)$

  - Example: [@clock $\geq 600$] covers [@clock $\geq 700$]

- $L_E(C)$: set of all elements that match an element filter $C$

- $C_1$ covers $C_2$, $C_1 \sqsupseteq C_2$ iff $L_E(C_1) \supseteq L_E(C_2)$

  - Example: [@clock $\geq 600$] covers [@manufacture = "AMD" $\land$ @clock $\geq 700$]

- $L_S(S)$: set of all elements that are selected by an element selector $S$

- $S_1$ covers $S_2$, $S_1 \sqsupseteq S_2$ iff $L_S(S_1) \supseteq L_S(S_2)$

- An absolute path covers another absolute path iff both are identical

- An abbreviated path covers another (abbreviated/absolute) path iff the former is a suffix of the later
1.3.2 Filter model III

- Example: //cpu covers //item/cpu
  - Because //cpu selects all elements named cpu, //item/cpu only selects those elements named cpu which are a sub-element of an element item

- \( L_P(P) \): set of all elements that match a path filter \( P \)

- \( P_1 = (S_1, C_1) \) covers \( P_2 = (S_2, C_2) \), \( P_1 \sqsupseteq P_2 \) iff \( L_P(P_1) \supseteq L_P(P_2) \)

- Example: //cpu[@manufacturer = “AMD”] covers //cpu[@manufacturer = “AMD” ∧ @clock ≥ 700]

- Lemma: Given two path filters \( P_1 = (S_1, C_1) \) and \( P_2 = (S_2, C_2) \):
  - \( P_1 \sqsupseteq P_2 \) iff \( S_1 \supseteq S_2 \land C_1 \supseteq C_2 \)
  - A filter \( F_1 \) covers \( F_2 \), \( F_1 \sqsupseteq F_2 \) iff \( N(F_1) \supseteq N(F_2) \)

- Lemma: Given two filters \( F_1 = P_1^1 \land ... \land P_n^1 \) and \( F_2 = P_1^2 \land ... \land P_m^2 \):
  - \( F_1 \sqsupseteq F_2 \) iff \( \forall i \exists j \) such that \( P_i^1 \sqsupseteq P_j^2 \)

- Example: the filter \{ //cpu[@type = “Athlon”] \} covers \{ //seller[@name = “Pu”] ∧ //cpu[@type = “Athlon” ∧ @clock ≥ 600] \}
1.3.2 Filter model IV

- \( S_1 \) is disjoint with \( S_2 \) iff \( L_S(S_1) \cap L_S(S_2) = \emptyset \)

- \( C_1 \) is disjoint with \( C_2 \) if there exists no attribute that is constrained in both element filters

  - Example: \([\@\text{minprice} \geq 100]\) is disjoint with \([\@\text{name} = \text{“Pu”}]\)

- \( P_1 \) is disjoint with \( P_2 \) iff \( S_1 \) is disjoint with \( S_2 \) or \( C_1 \) is disjoint with \( C_2 \)
1.4 Objects

- Model notifications and filters as objects
- Calling methods on attribute objects
  - Methods can be invoked on the objects embedded in the notification
  - The return value of the method can be a boolean value that is interpreted as a result of the attribute filter or a value that is used to evaluate the constraint

  **Example:** An instance of a class `StockQuote` has been embedded in a notification
  - The object possesses an attribute with the name `quote`
  - `A = (quote.id() = “IBM”)`
  - `A covers (quote.isRealTime()) \(\land\) (quote.id() = “IBM”) \(\land\) (quote.price() > 45.0)`
2 Distributed Event-Based Systems

2.1 Distributed system model
2.2 Distributed notification service architecture
2.3 Distributed notification routing
2.1 Distributed system model

Assumptions:

- Each node runs one or more processes
- Processes interact by passing messages via links between them
- A link connects a pair of processes and transmits messages asynchronously
- Message delay is unknown but finite
- A FIFO ordering of messages is applied
- System is not overloaded and fault-free (reliable processes and links)
The notification service forms an overlay network in the underlying system.

The overlay consists of event brokers that run as processes on physical nodes.

- Border and inner brokers forward the message to neighbouring brokers according to filter-based routing tables and routing strategies.
- Messages are sent to local brokers.
- Local brokers deliver the message to the application components.
2.3 Distributed notification routing

Routing =

- Matching of all the notifications with all the subscriptions
- Delivering of the notifications to all the clients and the neighbouring brokers with a matching subscription

1. Flooding:

- Brokers forward notifications to all the neighbouring brokers
- Only brokers to which subscribers are connected test on matching subscriptions
- Advantage: guarantee that all the notifications will reach their destination
- Drawback: many unnecessary messages are exchanged among brokers

2. Filter-based: depends on routing tables (RT), which are maintained by brokers

- A routing entry is a filter-destination pair \((F, D)\)
- Entries are updated by sending control messages

3. Gossiping (not studied here)
2.3.1 Strategy of filter-based routing

- **Simple routing:**
  - Each broker has global knowledge about all active subscriptions
    - Routing tables may grow excessively
    - High filter forwarding overhead if subscriptions change frequently

- **Advanced routing:**
  - **Identity-based routing:** avoids forwarding of a subscription that matches identical subscriptions
  - **Covering-based routing:** avoids forwarding of those subscriptions that only accept a subset of notifications matched by a previously forwarded subscription
  - **Merging-based routing:** the broker creates a new cover for the merged routing entries that replaces the old ones
3 Formal specification of distributed routing

3.1 Distributed system model for routing
3.2 Notations fornotif. forwarding and delivery
3.3 Valid routing
3.4 Safety and liveness conditions of valid routing
3.5 Safety and liveness conditions of monotone valid routing
3.1 Distributed system model for routing

Assumptions:

- Acyclic connected topologies
  - Can be circumvented by running a spanning tree algorithm on original (potentially cyclic) topology
- Topology is static and clients are stationary
- System without advertisements
3.2 Notations for notif. forwarding and delivery

- $T_B^D$: set of filters of the routing table of broker $B$ regarding single destination $D$
  $$T_B^D = \{F | \exists (F, D) \in T_B\}$$

- $T_B^{\setminus D}$: set of filters regarding all but single destination $D$
  $$T_B^{\setminus D} = \{F | \exists (F, E) \in T_B \land E \neq D\}$$

- $N(T_B^D)$: set of notifications that match $T_B^D$

- $N_B$: set of neighbouring brokers

- $L_B$: set of local consumers

- $F_B(n)$: destinations to which a broker $B$ forwards or delivers a notification $n$
  $$F_B(n) = \{D | D \in N_B \cup L_B \land n \in N(T_B^D)\}$$
3.3 Valid routing

- Valid routing algorithm = adapts the routing configuration by preserving the safety and liveness properties of the DEBS

- Additional notations:
  - \( \theta(Y) \): access broker — i.e. identity of the broker that manages consumer \( Y \)
  - Simple directed path connecting a broker with \( \theta(Y) \)
    - \( B_1, \ldots, B_j \) simple path in the network of brokers
    - \( \gamma(B_1, \ldots, B_j) \): set of notifications such that if a notification is published at \( B_j \) and stays in this set, it reaches \( B_1 \) over this path
      - \( \gamma(B_1, \ldots, B_j) = \bigcap_{1 < k \leq j} N(T|_{B_k}) \)
3.4 Safety and liveness conditions of valid routing

■ To guarantee safety, the local routing configuration ensures that only matching notifications are delivered

- **Local subset validity**

\[\Box \left[ N(T_{\theta(Y)}) \subseteq N(S_Y) \right] \quad (=r1)\]

■ To guarantee liveness, when a consumer \(Y\) subscribes to a filter \(F\) and stays subscribed, then from some time, every notification that is published at any broker \(B\) and that matches \(F\) should be delivered to \(Y\)

- **Eventual super-set validity**

\[\Box \left( (F \in S_Y) \implies \Diamond \Box \left[ N(T_{\theta(Y)}) \supseteq N(F) \right] \right) \quad (=r2: \text{From } \theta(Y) \text{ to } Y)\]

\[\Box \left( (F \in S_Y) \land B \neq \theta(Y) \land n \in N(F) \implies \Diamond \Box \left[ n \in \gamma(\theta(Y), \ldots, B) \right] \right) \quad (=r3: \text{From } B \text{ to } \theta(Y))\]
3.4.1 Monotone valid routing algorithms

- **Drawbacks of valid routing**
  - Local subset validity does not require immediate delivery
  - Eventual super-set validity is a property of the routing configuration of the entire topology

- **Improvements**
  - Immediate delivery
    - Local consumer subscription followed by local publisher publication should imply local notification of the consumer
  - Set of notifications forwarded is monotonically increasing for any path
    - Notifications sent over $B_{i+1} \rightarrow B_i$ are sent over $B_{i+2} \rightarrow B_{i+1}$
      - Only depends on the routing configurations of neighbouring brokers
3.5 Safety and liveness conditions of monotone valid routing

■ Reminder:

- $T^D_B = \{ F | \exists (F, D) \in T_B \}$
- $T^D_B = \{ F | \exists (F, E) \in T_B \land E \neq D \}$

■ Local validity $\equiv$ immediate delivery

$$\Box \left[ N(T^Y_{\theta(Y)}) = N(S_Y) \right] \quad (= \text{merging of r1 and r2 + strengthness})$$

■ Eventual monotone remote validity$^2$

$$\Box \left[ \Box [ n \in N(T^B_{Bj}) ] \implies \Diamond \Box [ n \in N(T^B_{Bi}) ] \right]$$

2. If $n$ is forwarded to $B_k \neq B_j \in N_B$ then $n$ comes from $B_j$. 
4 Distributed routing algorithms

4.1 Generic algorithm
4.2 Flooding
4.3 Simple routing
4.4 Identity-based routing
4.1 Generic algorithm

4.1.1 Main program
4.1.2 handleMessage procedure
4.1.3 handleNotification procedure
4.1.4 Preliminary words about the generic administer procedure
4.1.5 handleAdminMessage procedure
4.1.6 pub, sub and unsub procedures
4.1.1 Main program

- The main program starts when the broker is created:
  1. Initialise the routing table $T_B$ of the broker $B$
  2. Initialise a delivery queue $Q_C$ for each local consumer $C$
  3. Enter an infinite loop that dispatches messages arriving from neighbouring brokers to the `handleMessage` procedure

```
Program ContentBasedRouting()
    initialise $T_B$
    initialise $Q_C$ for all $C \in L_B$
    loop
      wait until a message is available
      $m \leftarrow$ next selected message
      `handleMessage($m$)`
```

Routing table of $B_1$

- $B_1$:
  - $(F_1, X_2)$
  - $(F_2, X_3)$
  - $(F_3, B_2)$
  - $(F_4, B_3)$
4.1.2 handleMessage procedure

- handleMessage dispatches a message based on message type
  - Two types of messages are exchanged among neighbouring brokers
    1. forward($n$): to disseminate a notification $n$ in the network of brokers
    2. admin($S$, $U$): to propagate routing table updates
      - $S$: set of subscriptions
      - $U$: set of unsubscriptions
    3. administer($S$, $U$): to compute the admin messages to send
      - $M_S$: set of pairs (filter$_{sub}$, destination) for sending admin messages
      - $M_U$: set of pairs (filter$_{unsub}$, destination) for sending admin messages

1. **procedure** handleMessage(Message $m$)
2. if $m$ is forward($n$) from neighbour $u$ then
3.   handleMessageNotification($u$, $n$)
4. if $m$ is admin($S$, $U$) from neigh. $u$ then
5.   $(M_S$, $M_U$) ← administer($u$, $S$, $U$)
6.   handleMessageAdminMessage($u$, $M_S$, $M_U$)
4.1.3 handleNotification procedure

- handleNotification sends forward messages to neighbouring brokers
- handleNotification notifies local consumers
  - notify is called by the broker to notify a local consumer about a notification
  - The notification is appended to the delivery queue $Q_Y$ of the consumer $Y$

1. **procedure**
   handleNotification(Neighbour $D$, Notification $n$)
2. send “forward($n$)” to all the neighbours $\in F_B(n) \setminus \{D\}$
3. **forall** local consumers $C \in F_B(n)$ do
4. notify($C$, $n$)
5. **procedure** notify(Consumer $Y$, Notification $n$)
6. $Q_Y \leftarrow$ append($Q_Y$, $n$)
The code of `administer` is implemented by framework instantiations to realise a concrete routing algorithm:

- Flooding
- Simple
- Identity-based
- Covering-based
- Perfect merging
- Imperfect merging

`administer` returns two sets that are pairs: `(filter\textsubscript{sub}, destination)` or `(filter\textsubscript{unsub}, destination)`

- Send an admin message to `destination` for `filter_{sub}`

  Sending done in `handleAdminMessage`, as explained in next slide.
4.1.5 **handleAdminMessage procedure**

- The values returned by `administer` are used as input to `handleAdminMessage`
- `handleAdminMessage` sends admin messages to neighbouring brokers

```plaintext
procedure handleAdminMessage(Dest D, Set MS, Set MU)
forall Bi ∈ NB \ {D}
S' ← \{F|(F, Bi) ∈ MS\}
U' ← \{F|(F, Bi) ∈ MU\}
if S' ≠ ∅ \lor U' ≠ ∅ then
send "admin(S', U')" to Bi
```
4.1.6 pub, sub and unsub procedures

- pub is called by a local publisher to publish a notification
- sub is called by a local consumer to subscribe to a filter
- unsub is called by a local consumer to unsubscribe to a filter

1. procedure pub (Publisher X, Notification n)
   handleNotification(X, n)
2. procedure sub (Consumer Y, Filter F)
   (M_S, M_U) ← administer(Y, {F}, ∅)
   handleAdminMessage(Y, M_S, M_U)
3. procedure unsub (Consumer Y, Filter F)
   (M_S, M_U) ← administer(Y, ∅, {F})
   handleAdminMessage(Y, M_S, M_U)
4.2 Flooding

- Idea: a broker forwards a notification to all its neighbours
  - Each broker is initialised to the set \(\{(F_T, U) | U \in N_B\}\) with \(\forall n \in N, F_T(n) = \text{true}\)

- Each broker updates its routing table (RT) regarding its local consumers
  - If a consumer \(Y\) subscribes to a filter \(F\), the broker adds \((F, Y)\) to its RT
  - If a consumer \(Y\) unsubscribes to a filter \(F\), the broker deletes \((F, Y)\) from its RT

- Flooding does not require the remote routing configuration to be updated

1. **procedure** `administer`(Dest \(s\), Set \(S\), Set \(U\))
2. \(T_B \leftarrow T_B \cup \{(F, s) | F \in S\}\)
3. \(T_B \leftarrow T_B \setminus \{(F, s) | F \in U\}\)
4. return \((\emptyset, \emptyset)\);
4.3 Simple routing

- Idea: use filter forwarding to update the routing configuration in reaction to subscribing and unsubscribing consumers

- Initially, $\forall B, T_B = \emptyset$

```plaintext
procedure administer(Dest D, Set S, Set U)

1. $T_B \leftarrow T_B \cup \{(F, D)|F \in S\}$
2. $T_B \leftarrow T_B \setminus \{(F, D)|F \in U\}$
3. $M_S \leftarrow \{(F, H)|H \in B \setminus \{D\} \land F \in S\}$
4. $M_U \leftarrow \{(F, H)|H \in B \setminus \{D\} \land F \in U\}$
5. return $(M_S, M_U)$

Routing table of $B_1$
```

Denis Conan

Content-Based Distributed Event-Based System
4.4 Identity-based routing

Reminder:

- $T_B^D = \{ F | \exists (F, D) \in T_B \} $
- $T_B^D = \{ F | \exists (F, E) \in T_B \land E \neq D \} $

Idea: a subscription (unsubscription) is only forwarded to a neighbour $H$ if there is no identical subscription in the RT for a destination distinct from $H$

The superscript stands for Identical

- $C_B^I(F, D)$: set of routing entries in $T_B$ of which the filter is identical to the filter $F$ and of which the destination equals the destination $D$
  
  $$ C_B^I(F, D) = \left\{ (G, D) | (G, D) \in T_B \land F \equiv G \right\} $$

- $D_B^I(F)$: set of neighbours $H$ for which there is no routing entry $(G, D)$ in $T_B$, where $G$ is identical to $F$ and $D$ is distinct from $H$
  
  $$ D_B^I(F) = \{ H \in N_B | \not\exists G \in T_B^H : F \equiv G \} $$
4.4.1 Algorithm

- If a broker $B$ receives a(n) (un)subscription from a neighbour or a consumer $D$:
  - $B$ updates its RT (lines 4-6):
    - If $D$ is a neighbour, $B$ removes $C_B^l(F, D)$ (line 5)
    - If $D$ is a local consumer, $B$ removes solely $(F, D)$ (line 6)
  - $B$ forwards $F$ to all neighbours that are in $D_B^l(F)$ except $D$ (lines 7–10 and 13)
  - If $F$ is a subscription, $B$ inserts a routing entry $(F, D)$ into its RT (line 11)

```plaintext
procedure administer(Dest $D$, Set $S$, Set $U$)
    $M_S \leftarrow \emptyset$;
    $M_U \leftarrow \emptyset$;
    for all $F \in S \cup U$ do
        if $D \in N_B$ then
            $T_B \leftarrow T_B \setminus C_B^l(F, D)$;
        else
            $T_B \leftarrow T_B \setminus (F, D)$;
        endif
    done
    $A \leftarrow \{(F, H) | H \in D_B^l(F) \setminus \{D\}\}$;
    if $F \in U$ then
        $M_U \leftarrow M_U \cup A$;
    else
        $M_S \leftarrow M_S \cup A$;
        $T_B \leftarrow T_B \cup \{(F, D)\}$;
    endif
    return $(M_S, M_U)$;
```

Denis Conan
Content-Based Distributed Event-Based System
5 Lab on content-based filtering with the muDEBS framework

- A result of our research on context data distribution for the Internet of Things
  - **muDEBS**
    - muDEBS uses muSCA
    - muDEBS is used by muContext
    - muDEBS is used by QoCIM

- We propose to follow a tutorial on **muDEBS** in order to see in action the generic algorithm in the case “simple routing”