Distributed Event-Based System — Addendum

Denis Conan

Septembre 2018

1 Formal specification of simple event-based system

1.1 Formal background — Temporal logic

- Trace: a sequence of states: $\sigma = s_0, s_1, s_2, \ldots$
  - Subtrace: $\sigma_{|i}$ is the trace $s_i, s_{i+1}, s_{i+2}, \ldots$
- Atomic predicate P is true for every trace whose first state satisfies P
- Formula Ψ: P with quantifiers ($\forall$, $\exists$) and logical operators ($\vee$, $\wedge$, $\implies$, $\neg$)
- Temporal operators:
  - $\square$ ("always")
    - $\square\Psi$ is true for traces σ iff $\forall i \geq 0, \Psi$ is true for $\sigma_{|i}$
    - $\square P$ means P always holds, *i.e.* for all subtraces
  - $\diamond$ ("eventually")
    - $\diamond\Psi$ is true for traces σ iff $\exists i \geq 0 : \Psi$ is true for $\sigma_{|i}$
    - $\diamond P$ means P will hold eventually, *i.e.* there is a subtrace for which P holds
  - $\bigcirc$ ("next")
    - $\bigcirc\Psi$ is true for traces σ iff Ψ is true for $\sigma_{|1}$
    - $\bigcirc P$ means P holds for the subtrace starting at the second place of the trace

1.1.1 Exercise

- ? $\Box\Diamond P$
- ? $\Diamond\Box P$
- ? $\Box[P \implies \Box P]$
- ? $\Box[P \implies \Diamond Q]$
- ? $\Box[P \implies \bigcirc\Box\neg P]$
- ? $P \implies \Diamond\Box Q$
- ? $\Box\neg P \vee \Box\neg Q \equiv \neg(\Diamond P \wedge \Diamond Q)$

1.2 Changes of the state caused by interface operations (w/o advertisements)

- X: a component of a system (being a producer and/or a consumer)
- $\mathcal{C}$: the set of all the components
- S_X: a set of active subscriptions for component X
- P_X: a set of published notifications by component X
- D_X: a set of delivered notifications to component X
- $\mathcal{N}$: the set of all the notifications, $N \subseteq \mathcal{N}$: a set of notifications
  - $n \in N(S_X)$: X has a subscription that matches $n \in \mathcal{N}$

$sub(X, F)$	Component X subscribes to filter F	$S'_X = S_X \cup \{F\}$
$unsub(X, F)$	Component X unsubscribes to filter F	$S'_X = S_X \setminus \{F\}$
$pub(X, n)$	Component X publishes n	$P'_X = P_X \cup \{n\}$
$notify(X, n)$	Component X is notified about n	$D'_X = D_X \cup \{n\}$

- « ' » indicates the state of a variable after the execution of the interface operation

1.2.1 Exercise

? $\Diamond notify(X, n)$

? $\Box \neg unsub(X, F)$

? $\Box[notify(X, n) \implies \bigcirc \Box \neg notify(X, n)]$

? $\Box[notify(X, n) \implies n \in N(S_X)]$

? $\Box[notify(Y, n) \implies n \in \cup_{X \in \mathcal{C}} P_X]$

1.3 Trace-based specification of simple event-based system (w/o advertisements)

- A component receives
  (a) only notifications it is currently subscribed to
  (b) only notifications that have previously been published
  (c) a notification at most once
  (d) all future notifications matching one of its active subscriptions

  - Safety: demands that "something irremediably bad" will never happen

$$\begin{aligned}\square\Big[\mathit{notify}(Y,n) \implies & [n \in N(S_Y)] & (=\text{a})\\ & \wedge [n \in \cup_{X\in\mathcal{C}} P_X] & (=\text{b})\\ & \wedge [\bigcirc\square\neg \mathit{notify}(Y,n)]\Big] & (=\text{c})\end{aligned}$$

  - Liveness: requires that "something good" will eventually happen

$$\square\Big[\square\big(F \in S_Y\big) \implies \Diamond\square\Big[\mathit{pub}(X,n) \wedge n \in N(F) \implies \Diamond \mathit{notify}(Y,n)\Big]\Big] \quad (=\text{d})$$

1.4 Changes of the state caused by the adding advertisements

- A_X: set of all active advertisements of component X
- $n \in N(A_X)$: X has an advertisement that matches $n \in \mathcal{N}$

$adv(X, F)$	Component X advertises filter F	$A'_X = A_X \cup \{F\}$
$unadv(X, F)$	Component X unadvertises filter F	$A'_X = A_X \setminus \{F\}$

1.5 Safety specification of simple event system with advertisements

(a) + (b) + (c) +

(e) If a notification is published that does not match any of the active advertisements of the publishing component, the notification should not be delivered to any component

$$\Box\Big[[notify(Y,n) \implies \bigcirc\Box\neg notify(Y,n)] \qquad (=\text{c})$$

$$\wedge\ [notify(Y,n) \implies n \in \cup_{X\in\mathcal{C}} P_X \cap N(S_Y)] \qquad (=\text{b,a})$$

$$\wedge\ [pub(X,n) \wedge n \notin N(A_X) \implies \Box\neg notify(Y,n)]\Big] \qquad (=\text{e})$$

1.6 Liveness specification of simple event system with advertisements

f1) If a client Y is always subscribed to F and a client X always advertises G

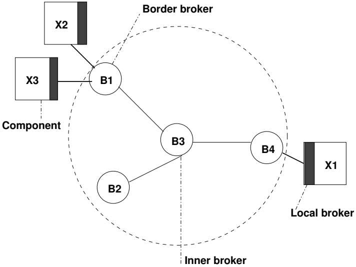
f2) then there exists a future time where a notification n published by X matches F and G

f3) will lead to the delivery of n to Y.

$$\begin{aligned}
\Box\Big[& [\Box(F \in S_Y) \wedge \Box(G \in A_X)] && (=\text{f1})\\
& \implies [\Diamond\Box(pub(X,n) \wedge n \in N(F) \cap N(G) && (=\text{f2})\\
& \implies \Diamond notify(Y,n))]\Big] && (=\text{f3})
\end{aligned}$$

2 Formal specification of distributed routing

2.1 Architecture of the distributed service



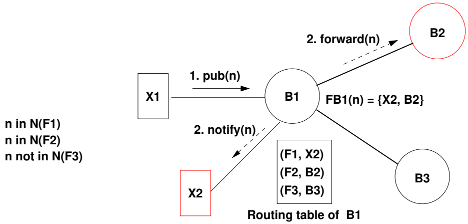
- The notification service forms an overlay network in the underlying system
- The overlay consists of event brokers that run as processes on nodes
  - Local brokers put the first message into the network
  - Border and inner brokers forward the message to neighbouring brokers according to filter-based routing tables and routing strategies
  - Messages are sent to local brokers
  - Local brokers deliver the message to the application components

2.2 Distributed system model for notification routing

- 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
- A FIFO ordering of messages is applied
- Acyclic connected topologies

- The topology of the overlay network of brokers is static
- Clients are stationary
- Communication channels are reliable and respect FIFO message ordering
- Message delay is unknown but finite, and system is not overloaded and fault-free
- System without advertisements

2.3 Notations for notification forwarding and delivery



- $T_B^{|D}$: set of filters of the routing table of broker B regarding single destination D
  $T_B^{|D} = \{F | \exists (F, D) \in T_B\}$
- $T_B^{\backslash D}$: set of filters regarding all but single destination D
  $T_B^{\backslash D} = \{F | \exists (F, E) \in T_B \wedge E \neq D\}$
- $N(T_B^{|D})$: set of notifications that match $T_B^{|D}$
- N_B: set of neighbouring brokers

2.4 Valid routing

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

- Additional notations:
  - $\theta(Y)$: identity of the broker that manages consumer Y
  - Simple directed path connecting a broker with $\theta(Y)$—i.e., the access broker
    - $B_1, ..., B_j$ simple path in the network of brokers
    - $\gamma(B_1, ..., B_j)$: 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
      - $\gamma(B_1, ..., B_j) = \bigcap_{1<k\leq j} N(T_{B_k}^{|B_{k-1}})$

2.5 Safety and liveness conditions of valid routing

- To guarantee safety, the local routing configuration ensures that only matching notifications are delivered
  - Local subset validity

$$\Box\left[N(T_{\theta(Y)}^{|Y}) \subseteq N(S_Y)\right] \qquad (=\text{r1})$$

- 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

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

$$\Box\Big[\Box(F \in S_Y) \wedge B \neq \theta(Y) \wedge n \in N(F)$$

$$\implies \Diamond\Box\left[n \in \gamma(\theta(Y), ..., B)\right]\Big] \qquad (=\text{r3: From } B \text{ to } \theta(Y))$$

2.6 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

2.7 Safety and liveness conditions of monotone valid routing

- Reminder:
  - $T_B^{|D} = \{F | \exists (F, D) \in T_B\}$
  - $T_B^{\setminus D} = \{F | \exists (F, E) \in T_B \wedge E \neq D\}$

- Local validity $\equiv$ immediate delivery

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

- Eventual monotone remote validity[1]

$$\Box\left[\Box\left[n \in N(T_{B_i}^{\setminus B_j})\right] \implies \Diamond\Box\left[n \in N(T_{B_j}^{|B_i})\right]\right]$$

1. If n is forwarded to $B_k \neq B_j \in N_{B_i}$ then n comes from B_j.

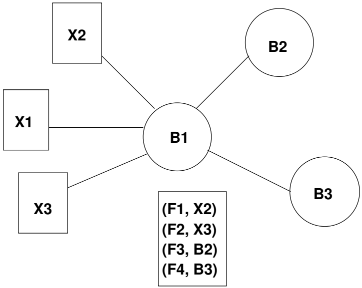
3 Routing algorithm framework

3.1 Generic algorithm

3.1.1 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)

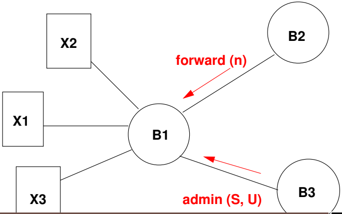


Routing table of B1

3.1.2 `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 $(\text{filter}_{\text{sub}}, \text{destination})$ for sending `admin` messages
       - M_S: set of pairs $(\text{filter}_{\text{unsub}}, \text{destination})$ for sending `admin` messages

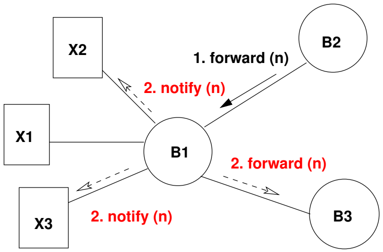
```
procedure handlemessage(Message m)
  if m is forward(n) from neighbour u then
    handleNotification(u, n)
  if m is admin(S, U) from neigh. u then
    (M_S, M_U) ← administer(u, S, U)
    handleAdminMessage(u, M_S, M_U)
```



3.1.3 `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

procedure handleNotification(Neighbour D, Notification n)
  send "forward(n)" to all the neighbours $\in F_B(n) \setminus \{D\}$
  forall local consumers $C \in F_B(n)$ **do**
    notify(C, n)
procedure notify(Consumer Y, Notification n)
  $Q_Y \leftarrow$ append(Q_Y, n)



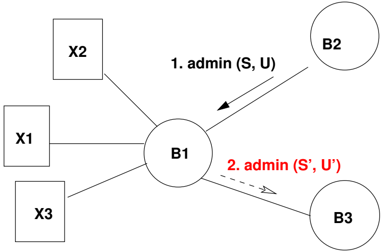
3.1.4 Preliminary words about the generic `administer` procedure

- 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: $(\text{filter}_{\text{sub}}, \text{destination})$ or $(\text{filter}_{\text{unsub}}, \text{destination})$
  - Send an `admin` message to *destination* for $\text{filter}_{\text{sub}}$
    - Sending done in `handleAdminMessage`, as explained in next slide

3.1.5 `handleAdminMessage` procedure

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

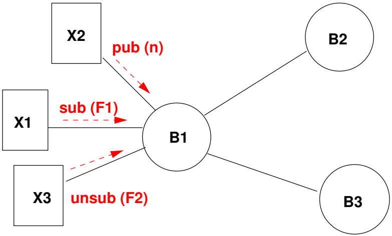
procedure handleAdminMessage(Dest D, Set M_S, Set M_U)
  forall $B_i \in N_B \backslash \{D\}$
    $S' \leftarrow \{F | (F, B_i) \in M_S\}$
    $U' \leftarrow \{F | (F, B_i) \in M_U\}$
    if $S' \neq \emptyset \vee U' \neq \emptyset$ **then**
      send "admin(S', U')" to B_i



3.1.6 `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

procedure pub (Publisher X, Notification n)
  handleNotification(X, n)
procedure sub (Consumer Y, Filter F)
  $(M_S, M_U) \leftarrow$ administer($Y, \{F\}, \emptyset$)
  handleAdminMessage(Y, M_S, M_U)
procedure unsub (Consumer Y, Filter F)
  $(M_S, M_U) \leftarrow$ administer($Y, \emptyset, \{F\}$)
  handleAdminMessage(Y, M_S, M_U)



3.2 Flooding

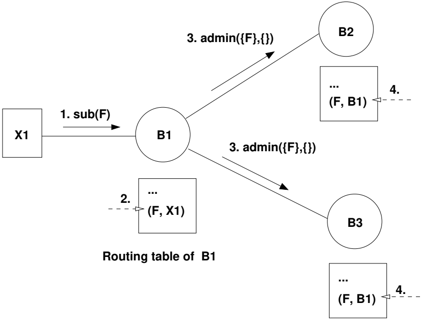
- Idea: a broker forwards a notification to all its neighbours
  - Each broker is initialised to the set $\{(F_T, U) | U \in N_B\}$ with $\forall n \in N, F_T(n) = true$
- Each broker updates its routing table (RT) regarding its local consumers
  - If a consumer Y subscribes to a filter F, the broker 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

procedure administer(Dest s, Set S, Set U)
  $T_B \leftarrow T_B \cup \{(F, s) | F \in S\}$
  $T_B \leftarrow T_B \setminus \{(F, s) | F \in U\}$
  return $(\emptyset, \emptyset)$;

3.3 Simple routing

- Idea: use filter forwarding to update the routing configuration in reaction to subscribing and unsubscribing consumers
  - Initially, $\forall B$, $T_B = \emptyset$

procedure administer(Dest D, Set S, Set U)
  $T_B \leftarrow T_B \cup \{(F, D) | F \in S\}$
  $T_B \leftarrow T_B \setminus \{(F, D) | F \in U\}$
  $M_S \leftarrow \{(F, H) | H \in N_B \setminus \{D\} \wedge F \in S\}$;
  $M_U \leftarrow \{(F, H) | H \in N_B \setminus \{D\} \wedge F \in U\}$;
  return (M_S, M_U);



3.4 Identity-based routing

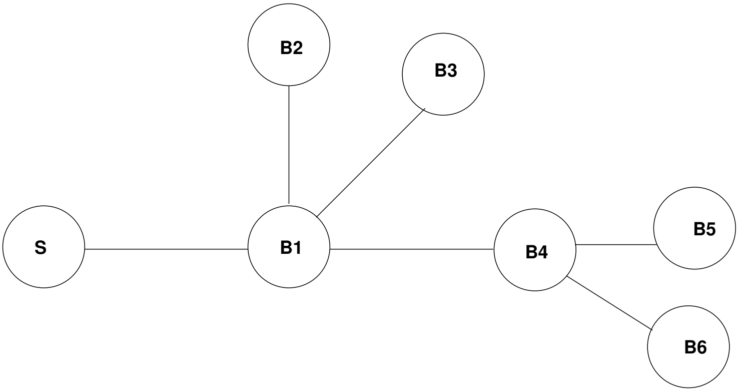
- Reminder:
  - $T_B^{|D} = \{F | \exists (F, D) \in T_B\}$
  - $T_B^{\setminus D} = \{F | \exists (F, E) \in T_B \wedge E \neq D\}$
- Idea: a subscription (unsubscription) is only forwarded to a neighbour H if there is no identical subscription in the RT for a destination distinct from H
- The superscript stands for *I*dentical
- $C_B^I(F, D)$: 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
  - $C_B^I(F, D) = \{(G, D) | (G, D) \in T_B \wedge F \equiv G\}$
- $D_B^I(F)$: 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
  - $D_B^I(F) = \{H \in N_B | \nexists G \in T_B^{\setminus H} : F \equiv G\}$

3.4.1 Algorithm

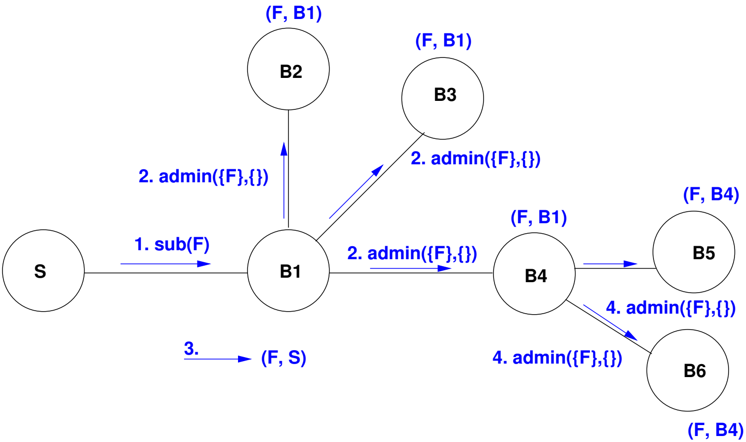
- If a broker B receives a(n) (un)subscription from a neighbour or a consumer D:
  - B updates its RT (lines 4-6):
    - If D is a neighbour, B removes $C_B^I(F, D)$ (line 5)
    - If D is a local consumer, B removes solely (F, D) (line 6)
  - B forwards F to all neighbours that are in $D_B^I(F)$ except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

1 **procedure** administer(Dest D, Set S, Set U)
2   $M_S \leftarrow \emptyset$;
3   $M_U \leftarrow \emptyset$;
4   **forall** $F \in S \cup U$ **do**
5   **if** $D \in N_B$ **then**
      $T_B \leftarrow T_B \setminus C_B^I(F, D)$;
6   **else** $T_B \leftarrow T_B \setminus (F, D)$;
7     $A \leftarrow \{(F, H) | H \in D_B^I(F) \setminus \{D\}\}$;
8   **if** $F \in U$ **then** $M_U \leftarrow M_U \cup A$;
9   **else**
10       $M_S \leftarrow M_S \cup A$;
11       $T_B \leftarrow T_B \cup \{(F, D)\}$;
12   **endif**
13   **return** (M_S, M_U);

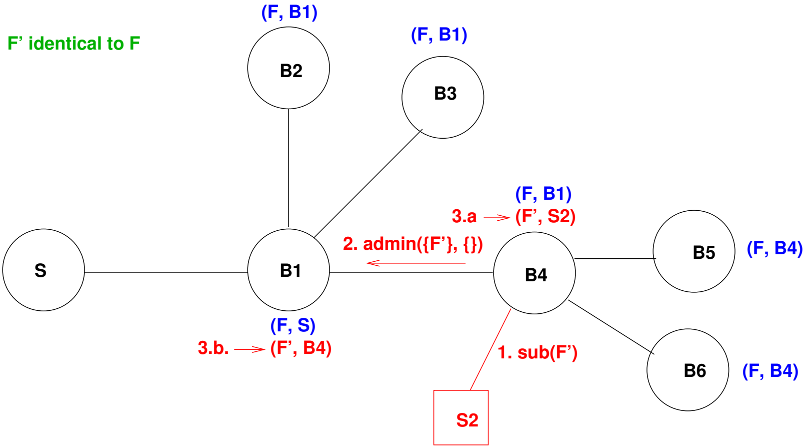
3.4.2 An example (1/3)



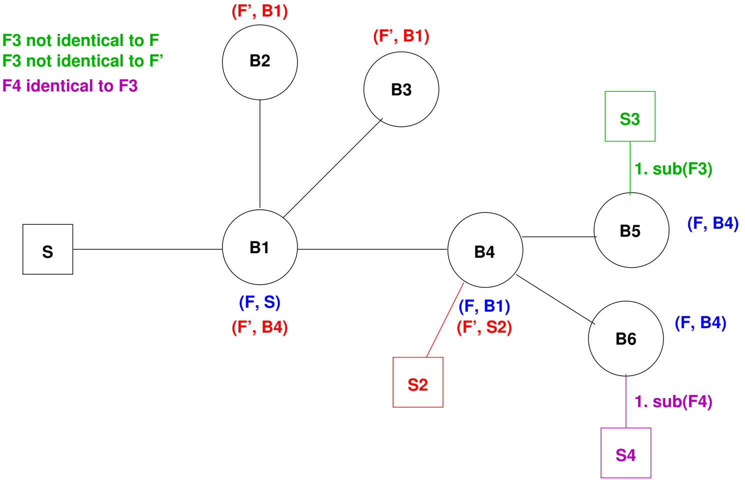
3.4.3 An example (2/3)



3.4.4 An example (3/3)



3.4.5 Exercise



? Execute the algorithm for the new subscription F_3 of S_3 and then the new subscription F_4 of S_4

4 Content-based data and filter models

4.1 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

4.2 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 (StockQuote,“Foo Inc”, 45) is matched by the subscription template (StockQuote,“Foo Inc”, *)

– Tuples with templates provide a simple model that is not flexible
  - Because attributes cannot be optional

4.3 Structured records

4.3.1 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: {(type, StockQuote),(name,"Infineon"),(price, 45.0)}

- More powerful than tuples since attributes can be optional in subscriptions and notifications

4.3.2 Filter model

- Attribute filter: triple $A_i = (n_i, Op_i, C_i)$
  with n_i = attribute name, Op_i = test operator, C_i = value for the test
- $L_A(A_i)$ = set of values v_i that cause an attribute filter to match attribute n_i
  - $L_A(A_i) = \{v_i | Op_i(v_i, C_i) = \mathit{true}\}$
  - Usually $L_A(A_i) \neq \emptyset$
- Filter F = boolean function applied to a notification n: $F(n) \rightarrow \{\mathit{true}, \mathit{false}\}$
- The set of matching notifications $N(F) = \{n | F(n) = \mathit{true}\} \subseteq \mathcal{N}$
- Simple filter = filter consisting of a single atomic predicate
- Compound filter = conjunction of simple filters: $F = A_1 \wedge ... \wedge A_n$
  - *E.g.*, (type = StockQuote) $\wedge$ (name = "Foo Inc") $\wedge$ (price $\notin [30, 40]$)
- A notification n matches a filter F *iff* it satisfies all the attributes filters of F

+ Attributes can be optional in the notification
+ New attributes can be added without affecting existing filters

4.3.3 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.*, $(price \in \{20, 21, 22, 23, 24, 25\})$ is identical to $(price \in [20, 25])$
- Overlapping:
  - $A_1 \sqcap A_2 \; iff \; n_1 = n_2 \land L_A(A_1) \cap L_A(A_2) \neq \emptyset$
  - *E.g.*, $(price > 25)$ overlaps $(price \in [20, 30])$
- Covering:
  - $A_1 \sqsupseteq A_2 \; iff \; n_1 = n_2 \land L_A(A_1) \supseteq L_A(A_2)$
  - *E.g.*, $A_1 = (price > 10)$ covers $A_2 = (price \in [20, 30])$
- Disjoint
  - $A_1 \not\sqcap A_2 \; iff \; n_1 = n_2 \land L_A(A_1) \cap L_A(A_2) = \emptyset$
  - $\{price < 10\}$ and $\{price > 20\}$ are disjoint

4.3.4 Routing optimisations with identity

- 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 $F_1 = A_1^1 \wedge ... \wedge A_n^1$ and $F_2 = A_1^2 \wedge ... \wedge A_m^2$ 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 $\wedge$ $(\forall i, \exists j \ : A_i^1 \equiv A_j^2)$

- *E.g.*, $F_1 = \{x = 4\} \wedge \{y > 5\}$ not identical to
  $F_2 = \{x = 4\} \wedge \{y > 5\} \wedge \{z \in [3, 5]\}$

4.3.5 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
  $\wedge$ to get rid of the obsolete[2] routing entries.

- Given Two filters $F_1 = A_1^1 \wedge \ldots \wedge A_n^1$ and $F_2 = A_1^2 \wedge \ldots \wedge A_m^2$ that are conjunctions of attribute filters with at most one attribute filter per attribute,
  $F_1 \sqsupseteq F_2$ *iff* $\forall i, \exists j \ : A_i^1 \sqsupseteq A_j^2$

- *E.g.*, $F_1 = \{x = 4\} \wedge \{y > 5\}$ covers $F_2 = \{x = 4\} \wedge \{y > 5\} \wedge \{z \in [3, 5]\}$

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

2. A routing entry covers another routing entry, which becomes obsolete

4.3.6 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

A_1	A_2	$A_1 \sqsupseteq A_2$ *iff*
n *instanceof* T_1	n *instanceof* T_2	$T_1 = T_2 \vee T_1$ *supertypeof* T_2

- Covering among comparison constraints on simple values

A_1	A_2	$A_1 \sqsupseteq A_2$ *iff*
$x \neq c_1$	$x < c_2$	$c_1 \geq c_2$
$x > c_1$	$x > c_2$	$c_1 \leq c_2$

  - *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 $A_2 = (x > 20) \implies A_1 \sqsupseteq A_2$

4.3.7 Covering with intervals and strings

- $n_1 = n_2$
- Covering among interval constraints on simple values

A_1	A_2	$A_1 \sqsupseteq A_2$ *iff*
$x \in I_1$	$x \in I_2$	$I_1 \supseteq I_2$
$x \notin I_1$	$x \notin I_2$	$I_1 \subseteq I_2$

  - 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

A_1	A_2	$A_1 \sqsupseteq A_2$ *iff*
s *hasPrefix* S_1	s *hasPrefix* S_2	S_2 *hasPrefix* S_1
s *hasPostfix* S_1	s *hasPostfix* S_2	S_2 *hasPostfix* S_1
s *hasSubstring* S_1	s *hasSubstring* S_2	S_2 *hasSubstring* S_1

  - E.g. $A_1 = (s$ *hasPrefix* "*abc*"$)$ and $A_2 = (s$ *hasPrefix* "*abcd*"$) \implies A_1 \sqsupseteq A_2$

4.3.8 Covering with sets

- $n_1 = n_2$
- Covering among set constraints on simple values

A_1	A_2	$A_1 \sqsupseteq A_2$ *iff*
$x \in M_1$	$x \in M_2$	$M_1 \supseteq M_2$
$x \notin M_1$	$x \notin M_2$	$M_1 \subseteq M_2$

- Covering among set constraints on multi values

A_1	A_2	$A_1 \sqsupseteq A_2$ *iff*
X *subset* M_1	X *subset* M_2	M_1 *superset* M_2
X *contains* a_1	X *superset* M_2	$a_1 \in M_2$
X *superset* M_1	X *superset* M_2	M_1 *subset* M_2
X *notContains* a_1	X *disjunct* M_2	$a_1 \in M_2$
X *disjunct* M_1	X *disjunct* M_2	M_1 *subset* M_2
X *overlaps* M_1	X *overlaps* M_2	M_1 *superset* M_2

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

4.3.10 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:
    - $F_1 = \{x = 5\} \wedge \{y \in \{2, 3\}\}$ and $F_2 = \{x = 5\} \wedge \{y \in \{4, 5\}\}$ can be merged to $F = \{x = 5\} \wedge \{y \in \{2, 3, 4, 5\}\}$
    - $F_1 = \{y = 3\} \wedge \{x = 5\}$ and $F_2 = \{y = 3\} \wedge \{x \neq 5\}$ can be merged to $F = \{y = 3\}$
  - Example of PERFECT merging rules for attribute filters

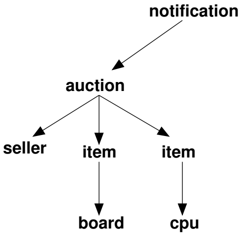
A_1	A_2	Condition	$A_1 \cup A_2$
$x \in M_1$	$x \in M_2$	-	$x \in M_1 \cup M_2$
$x \notin M_1$	$x \notin M_2$	$M_1 \cap M_2 = \emptyset$ $M_1 \cap M_2 \neq \emptyset$	$\exists x$ (i.e., no att. filter) $x \notin M_1 \cap M_2$
X *overlaps* M_1	X *overlaps* M_2	-	X *overlaps* $M_1 \cup M_2$
X *disjunct* M_1	X *disjunct* M_2	$M_1 \cap M_2 = \emptyset$	$\exists X$ (i.e., no att. filter)

4.4 Semi-structured records

4.4.1 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 $\Longrightarrow$ names address sets of attr.

```
<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>
```



4.4.2 Filter model I

- A filter model uses a path expression (e.g., XPath)
  - Select a set of attributes and Impose constraints on the selected attributes
- A filter is a conjunction of path filters: $F = \wedge_i P_i$
- 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: $C = \wedge_i A_i$
  - e.g. [@manufacturer = "AMD" $\wedge$ @clock $\geq$ 700]
- Example of path filter: /notification/auction/item/cpu[@manufacturer = "AMD" $\wedge$ @clock $\geq$ 700]

4.4.2 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* $n_1 = n_2 \wedge L_A(A_1) \supseteq L_A(A_2)$
  - 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" $\wedge$ @clock $\geq$ 700]
- C_1 is disjoint with C_2 if there exists no attribute that is constrained in both element filters
  - Example: [@minprice ¡ 100] is disjoint with [@name = "Pu"]
- $L_S(S)$: set of all elements that are selected by an element selector S
- S_1 covers S_2, $S_1 \sqsupseteq S_2$ *iff* $L_S(S_1) \supseteq L_S(S_2)$

4.4.2 Filter model III

- S_1 is disjoint with S_2 *iff* $L_S(S_1) \cap L_S(S_2) = \emptyset$
- 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
  - 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" $\wedge$ @clock $\geq$ 700]
- P_1 is disjoint with P_2 *iff* S_1 is disjoint with S_2 or C_1 is disjoint with C_2

4.4.2 Filter model IV

- Lemma: Given two path filters $P_1 = (S_1, C_1)$ and $P_2 = (S_2, C_2)$:
  $P_1 \sqsupseteq P_2$ *iff* $S_1 \sqsupseteq S_2 \wedge C_1 \sqsupseteq C_2$
  A filter F_1 covers F_2, $F_1 \sqsupseteq F_2$ *iff* $N(F_1) \supseteq N(F_2)$
- Lemma: Given two filters $F_1 = P_1^1 \wedge \ldots \wedge P_n^1$ and $F_2 = P_1^2 \wedge \ldots \wedge P_m^2$:
  $F_1 \sqsupseteq F_2$ *iff* $\forall i \exists j$ such that $P_i^1 \sqsupseteq P_j^2$
  - Example: the filter {//cpu[@type = "Athlon"]} covers
    {//seller[@name = "Pu"] $\wedge$ //cpu[@type = "Athlon" $\wedge$ @clock $\geq$ 600]}

4.5 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*
      - $A = (quote.id() = \text{“IBM”})$
      - A covers $(quote.isRealTime()) \wedge (quote.id() = \text{“IBM”})$ $\wedge (quote.price() > 45.0))$