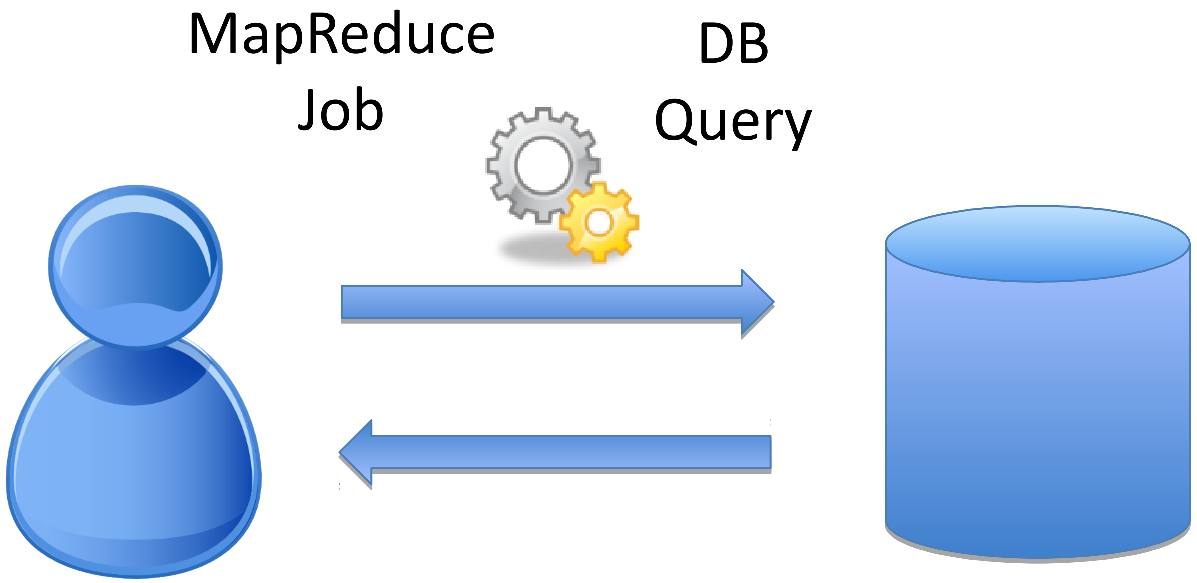
Event/Stream Processing

Alessandro Margara
Politecnico di Milano

Batch processing



Reactive applications

Financial Analysis

Traffic Monitoring

Fraud Detection

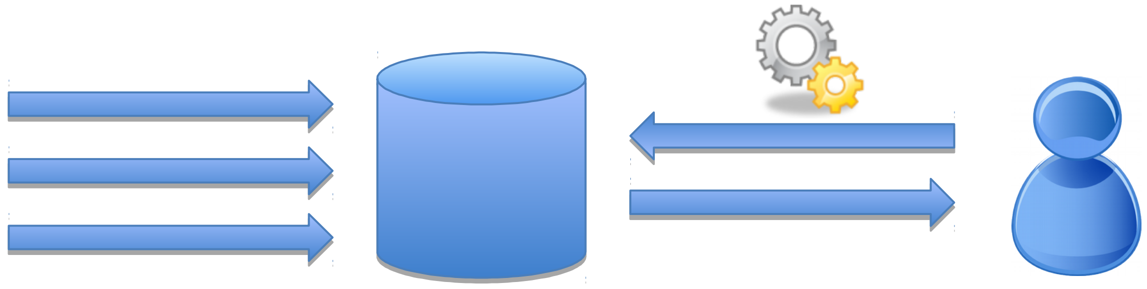
System Monitoring

Velocity!

Reactive applications

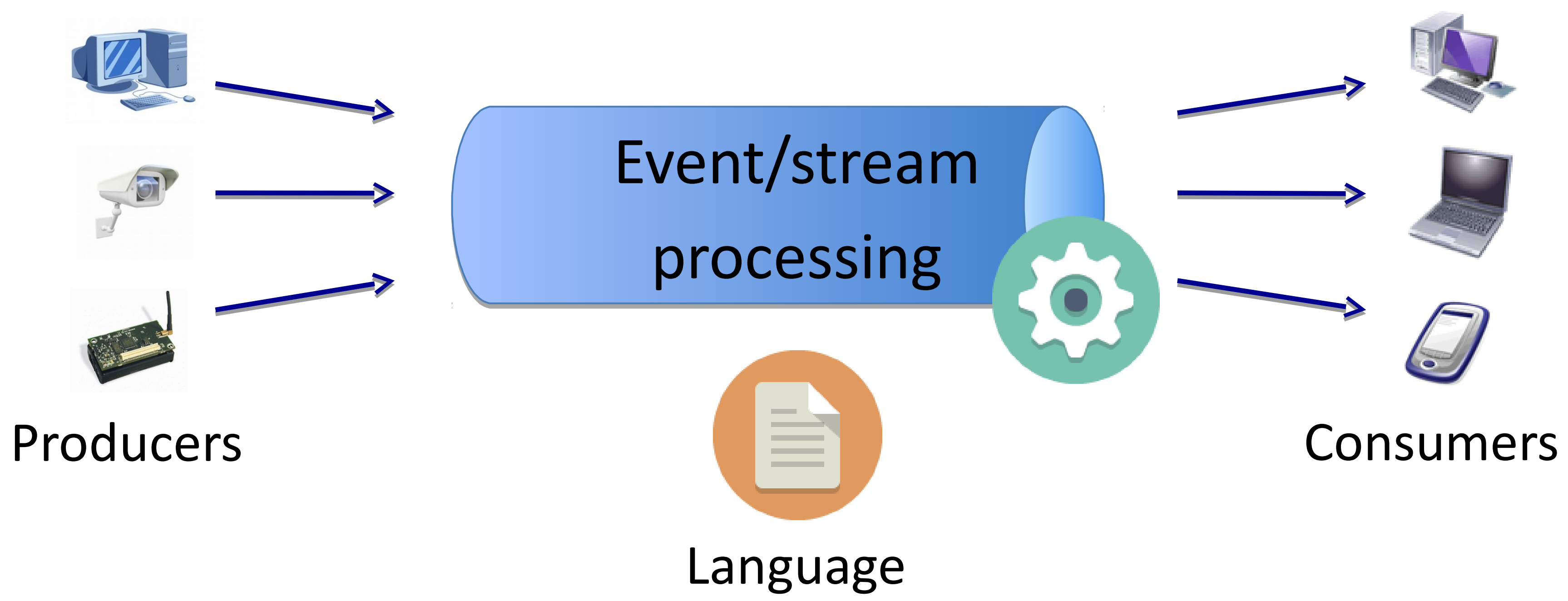
- Typical requirements
  - Process large volumes of data as soon as the data is produced …
    - High throughput
  - ... to timely produce new results
    - Low delay

Reactive applications



- Can we use existing technologies for batch processing?
  - They are not designed to minimize latency
  - We need a whole new model!

Event/stream processing



Language

- The language needs to provide suitable abstractions to capture the key elements of reactive, event-driven applications
  - Time / temporal relations
  - Seems pretty easy …
  - ... I'll try to convince you it is not ⏪

Processing

- Efficient algorithms to achieve
  - High throughput
  - Low delay
- Exploit parallel/distributed infrastructures
- Optimize processing and communication in distributed environments

Outline

- Background
- Esper: hands on
- Model

BACKGROUND

Background

- Active DBs
  - Early 90s
- Data Stream Management Systems (DSMSs)
  - 2000s
- Complex Event Processing (CEP)
  - 2000s
- Reactive Programming (RP)
  - Late 90s
  - Last few years

Active DB

- Traditional DB
  - Human-active database-passive
  - Processing is exclusively driven by queries

- Active DB
  - Event Condition Action (ECA) rules
  - Part of the reactive behavior moves from the application to the DB
  - Mostly DB extensions
    - View maintenance
    - Integrity checking

DSMS

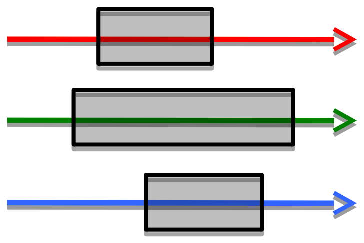
- Data streams are (unbounded) sequences of data elements

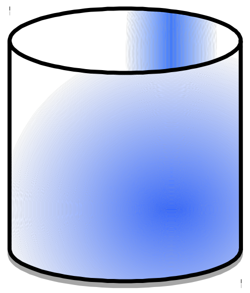
- Often, the most recent data is more relevant as it describes the current state of a dynamic system

DSMS

DBMS	**DSMS**
Persistent data	Transient streams
One-time queries	Continuous queries
Read intensive	Update intensive (append)
Random access	Sequential access (one pass)
Access plan determined based on the actual data	Unpredictable data characteristics and arrival patterns

DSMS (CQL)





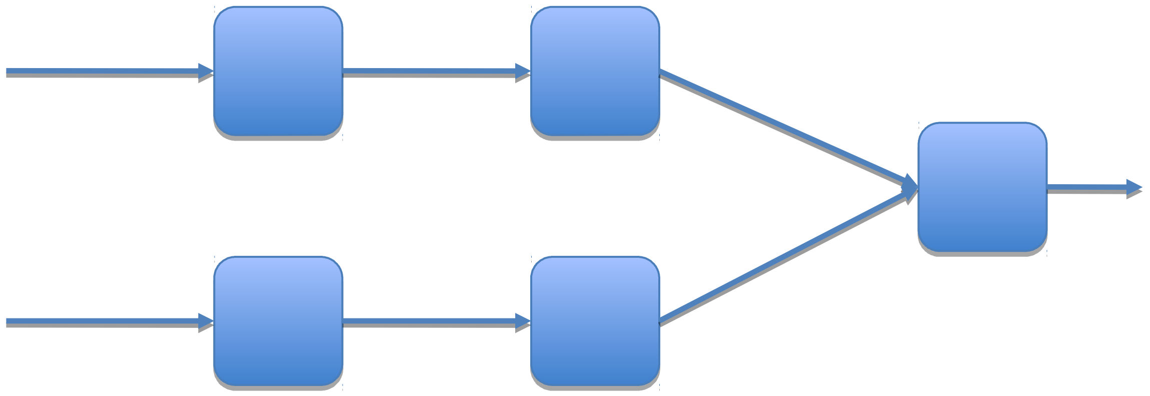
Stream-to-Relation
(Windows)

Relation-to-Relation
(Relational Operators)

Relation-to-Stream
(New/All results)

DSMS (SQuAl)

- Stream-to-stream operators
  - E.g., filter, project, map, aggregate, join, …
- Embedded windows to make operators non-blocking
- Operators combined in a dataflow graph

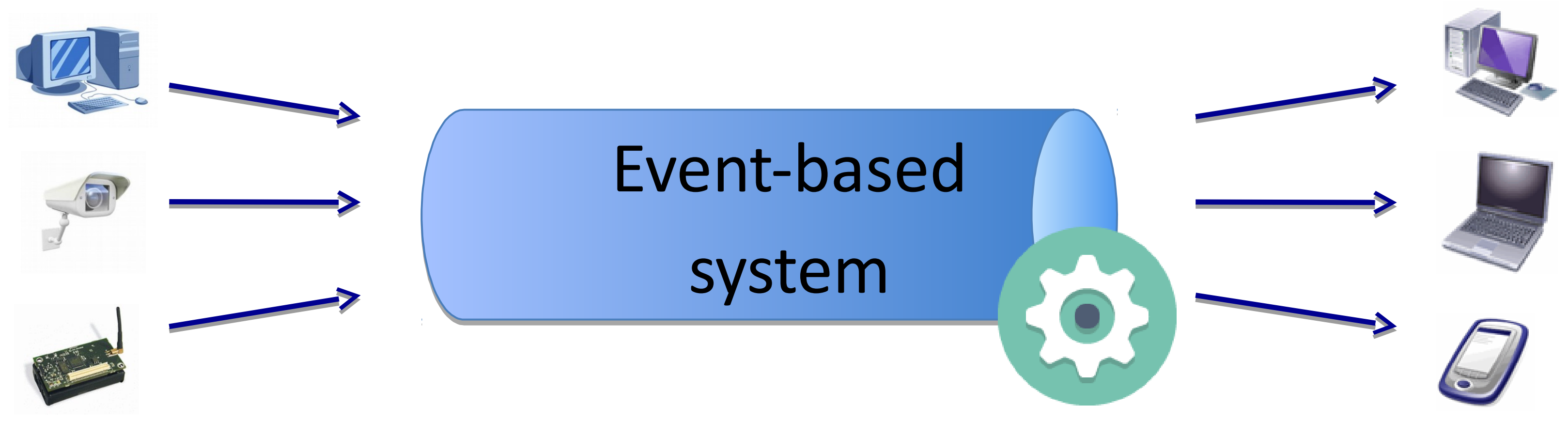


Event-based systems

- Software architecture in which the components
  - *Publish* notifications of event occurrences
  - *Subscribe* to the events they are interested in
- Ideal for dynamic environments
  - Loosely coupled components
  - Implicit communication
    - Anonymous
    - Asynchronous
    - Multicast

Event-based systems

- In event-based systems the processing task consists in *matching* events against subscriptions
- Different degrees of expressivity
  - topic-based, content-based, …

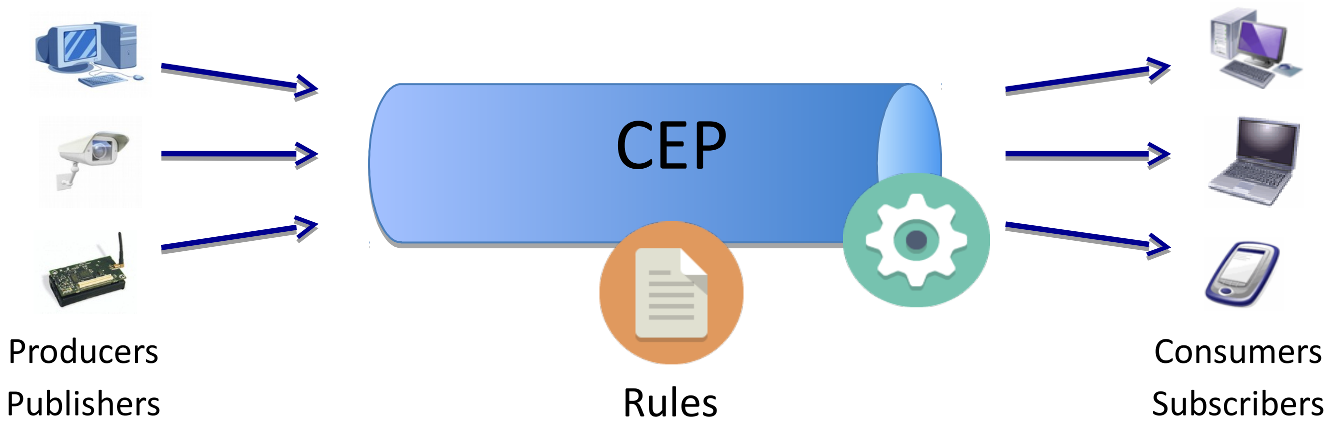


Producers
Publishers

Consumers
Subscribers

CEP

- CEP adds the ability to deploy rules that define *composite* events starting from *primitive* ones
  - E.g. if Temp(val > 10) and then Smoke within 5 min, trigger Fire



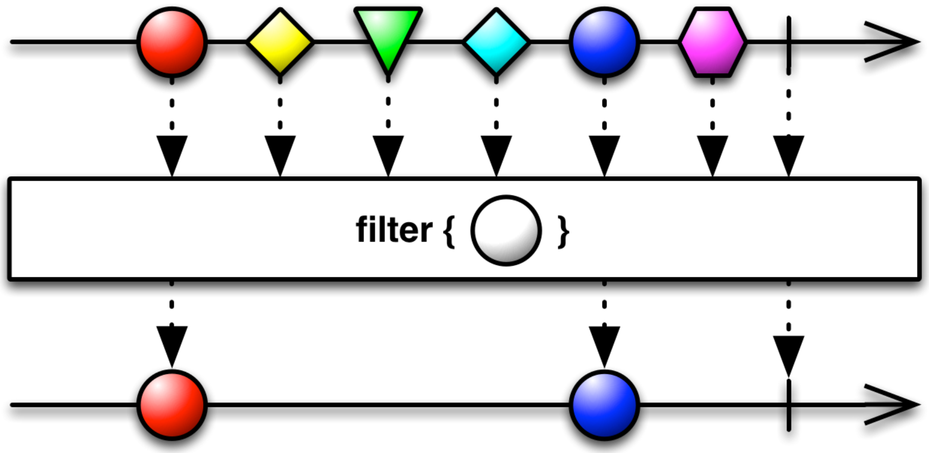
RP

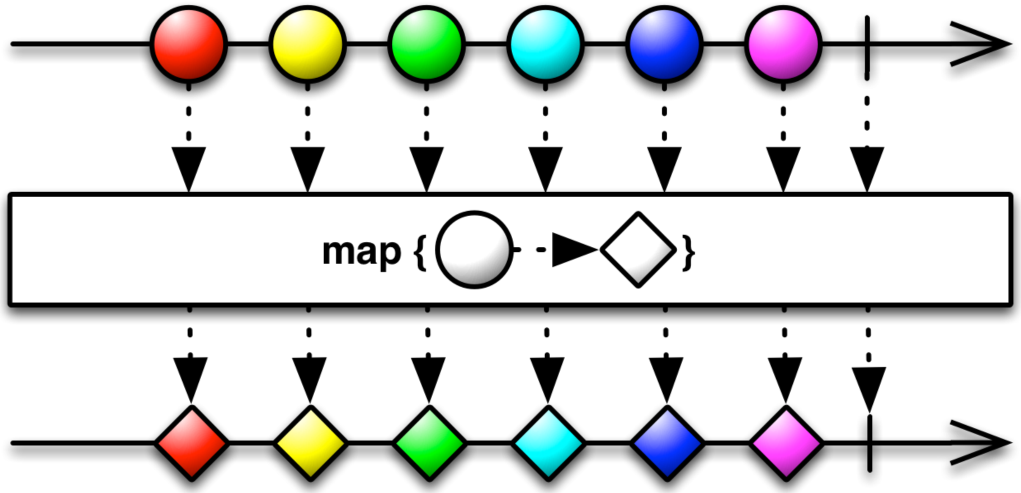
- Programming abstractions to simplify the design of reactive applications

- Focus on streams as unbounded collections of elements
  - (Functional) operators produce output streams from input streams
  - Similar to dataflow DSMSs

- Focus on programming language integration

RP





Big data + streaming = fast data

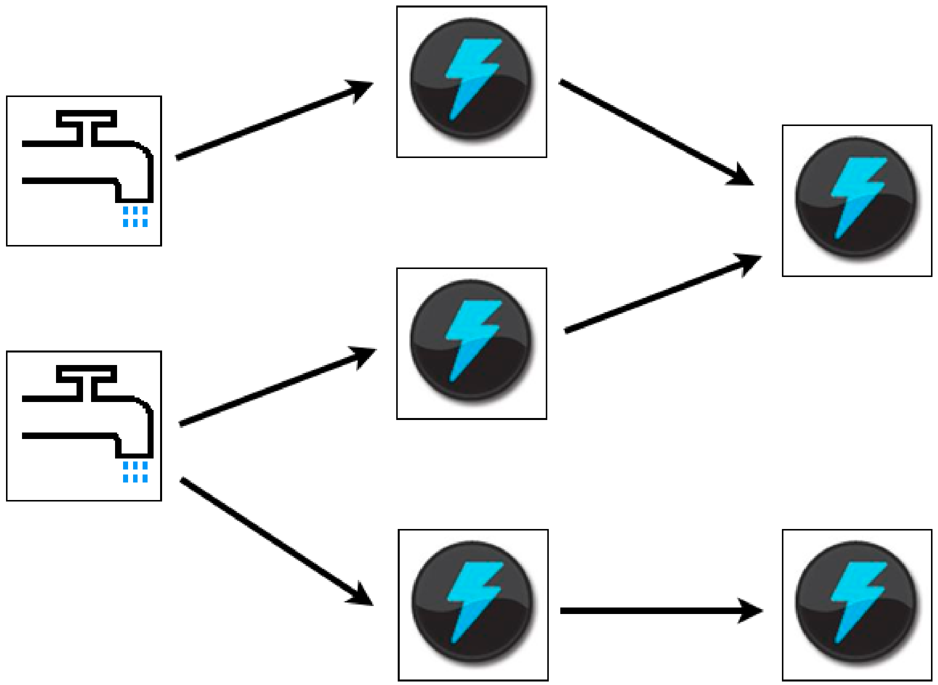
- Several systems have been proposed to perform streaming computations on clusters
  - Similar to MapReduce / Hadoop …
  - ... but focusing on streaming data

- Perhaps the most well known are
  - Apache Storm / Heron
    - Dataflow approach
    - Used within Twitter
  - Apache Spark Streaming, Apache Flink
    - Functional approach
    - You will see it in the next lectures

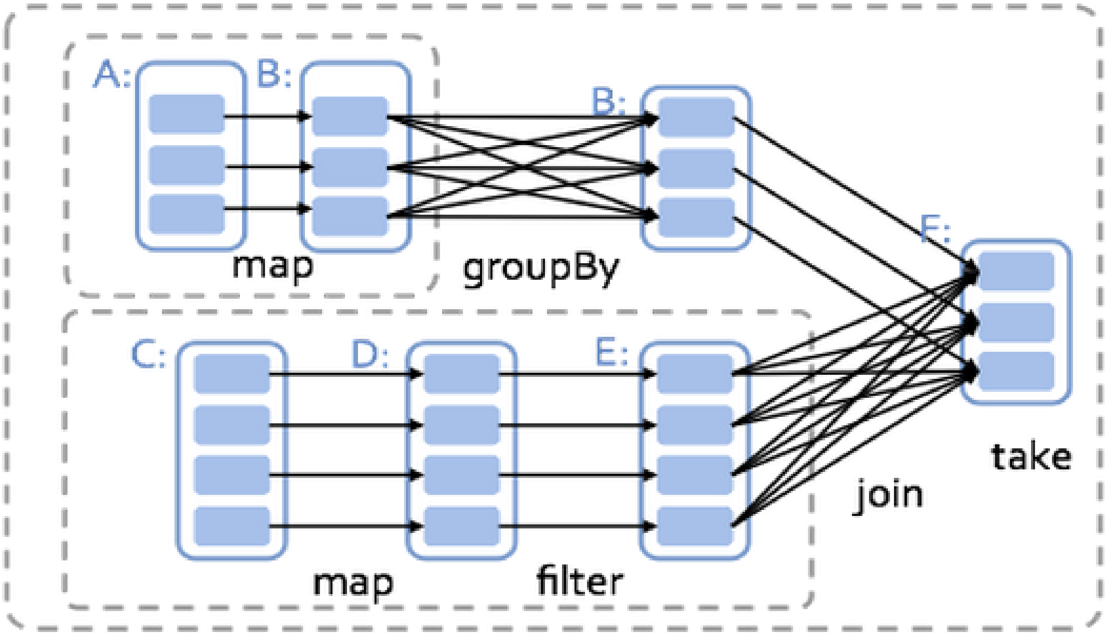
Big data + streaming = fast data

APACHE STORM™

Distributed • Resilient • Real-time



Big data + streaming = fast data



Big data + streaming = fast data

- New concerns
  - Query deployment in large computational infrastructures
    - Operator placement
    - Operator migration
  - Fault tolerance

ESPER

Esper in a nutshell

- EPL: rich language to express rules
  - Grounded on the DSMS approach
    - Windowing
    - Relational select, join, aggregate, …
    - Relation-to-stream operators to produce output
    - Sub-queries
  - Queries can be combined to form a graph
  - Introduces some features of CEP languages
    - Pattern detection
- Designed for performance
  - High throughput
  - Low latency

Esper in a nutshell

- Interaction with static / historical data
- Configurable push or pull communication
- Several adapters for input/output
  - CSV, JMS in/out, API, DB, Socket, HTTP
- Two versions
  - Esper ✉ Java
  - NEsper ✉ .NET / C#
- Esper HA
  - High Availability
  - Ensures that the state is recoverable in the case of failure

Running example

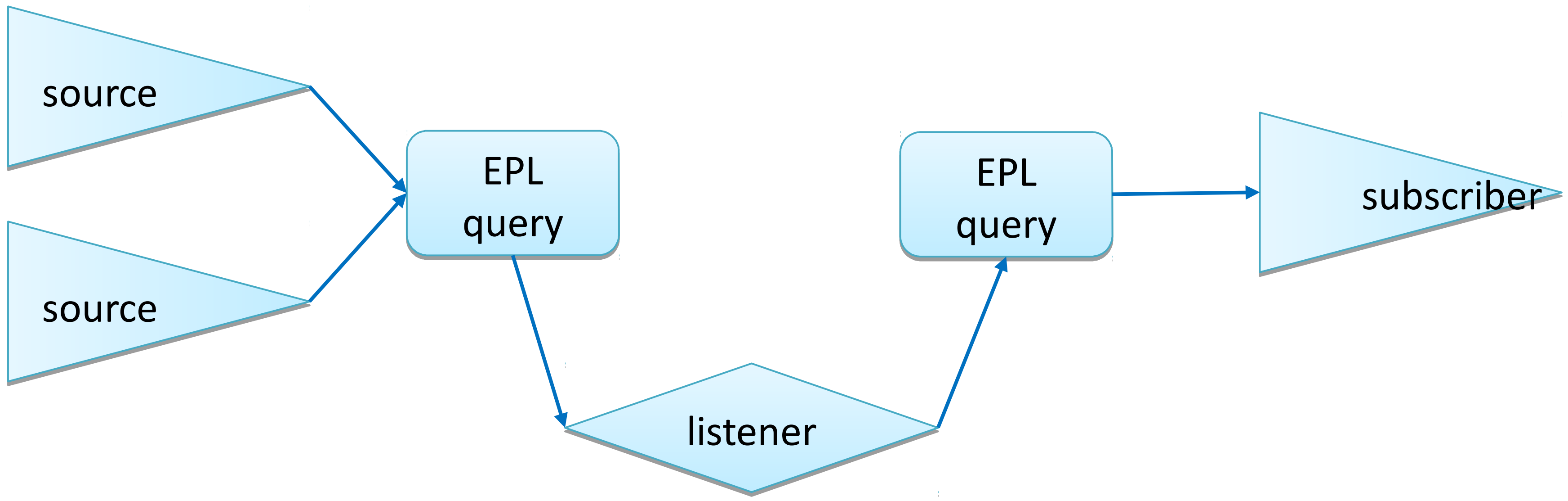
- Count the number of fires detected using a set of smoke and temperature sensors in the last 10 minutes
- Events
  - Smoke event: String sensor, boolean state
  - Temperature event: String sensor, double temperature
  - Fire event: String sensor, boolean smoke, double temperature
- Condition:
  - Fire: at the same sensor smoke followed by temperature>50

Processing model

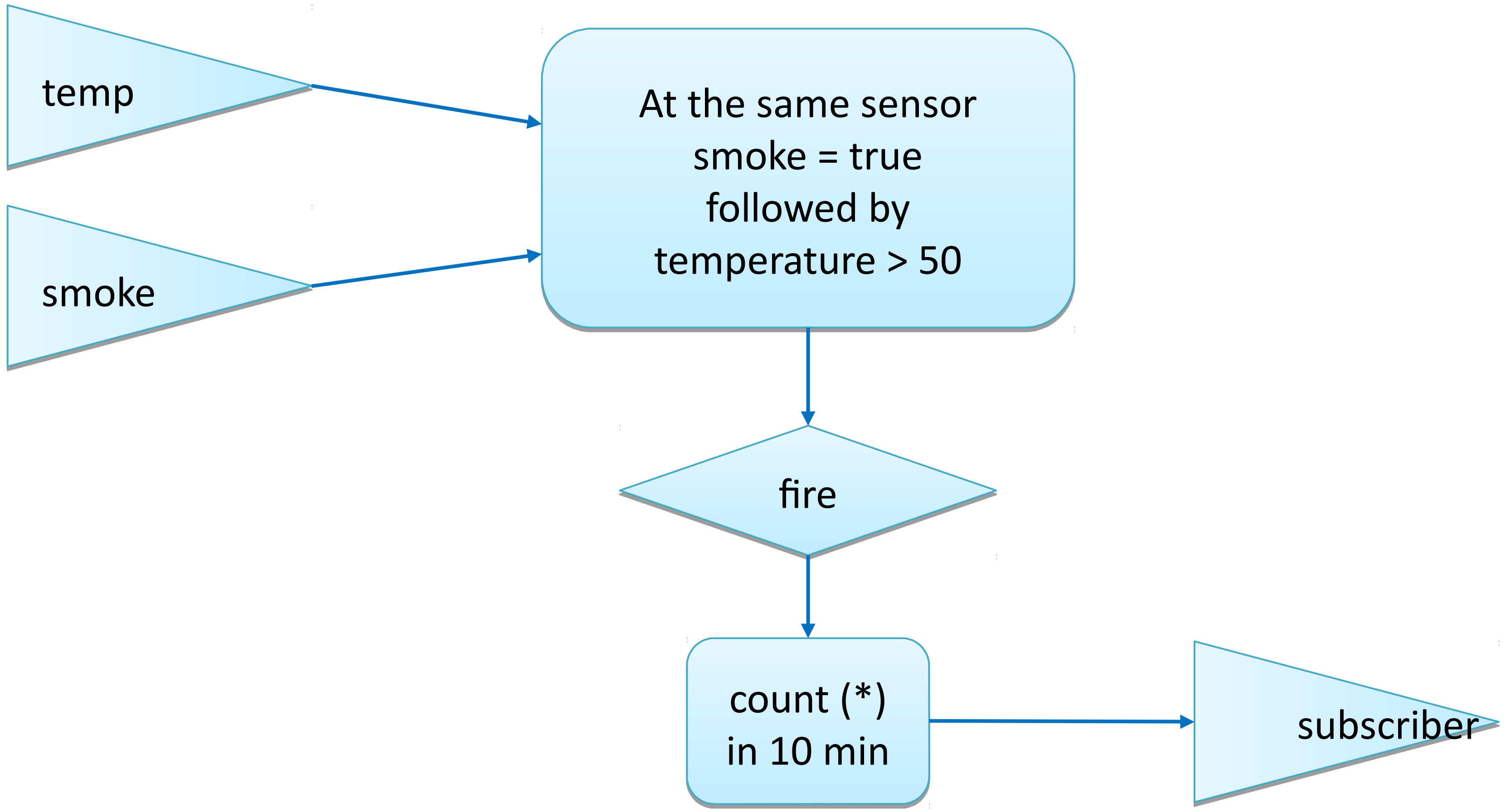
- Builds on four abstractions
  - Sources
    - Produce data items from sensors, trace files, etc.
  - Registered EPL queries
    - Continuously executed against the data items produced by the sources
  - Listeners
    - Receive data items from queries
    - Push data items to other queries
  - Subscribers
    - Receive processed data tuples

Processing model

- Sources, queries, listeners, and subscribers are connected to form a processing graph



Running Example



Declare event types

- Two ways
  - EPL *create schema clause*
  - Runtime configuration API *addEventType*

- Syntax

```
create schema
schema_name [as]
(property_name property_type
[,property_name property_type [,...])
[inherits inherited_event_type
[, inherited_event_type] [,...]]
```

Running example

```
create schema
   SmokeSensorEvent(
   sensor string,
   smoke boolean
);
```

```
create schema
   TemperatureSensorEvent(
   sensor string,
   temperature double
);
```

```
create schema
   FireComplexEvent(
   sensor string,
   smoke boolean,
   temperature double
);
```

Event Processing Language (EPL)

- EPL is similar to SQL
  - Select, where, …

- Event streams and views instead of tables
  - Views define the data available for the query
  - Views can represent windows over streams
  - Views can also sort events, derive statistics from event attributes, group events, …

EPL syntax

```
[insert into insert_into_def]
select select_list
from stream_def [as name]
[, stream_def [as name]] [,...]
[where search_conditions]
[group by grouping_expression_list]
[having grouping_search_conditions]
[output output_specification]
[order by order_by_expression_list]
[limit num_rows]
```

Simple examples

```
select   *
from     TemperatureSensorEvent
where    temperature>50
```

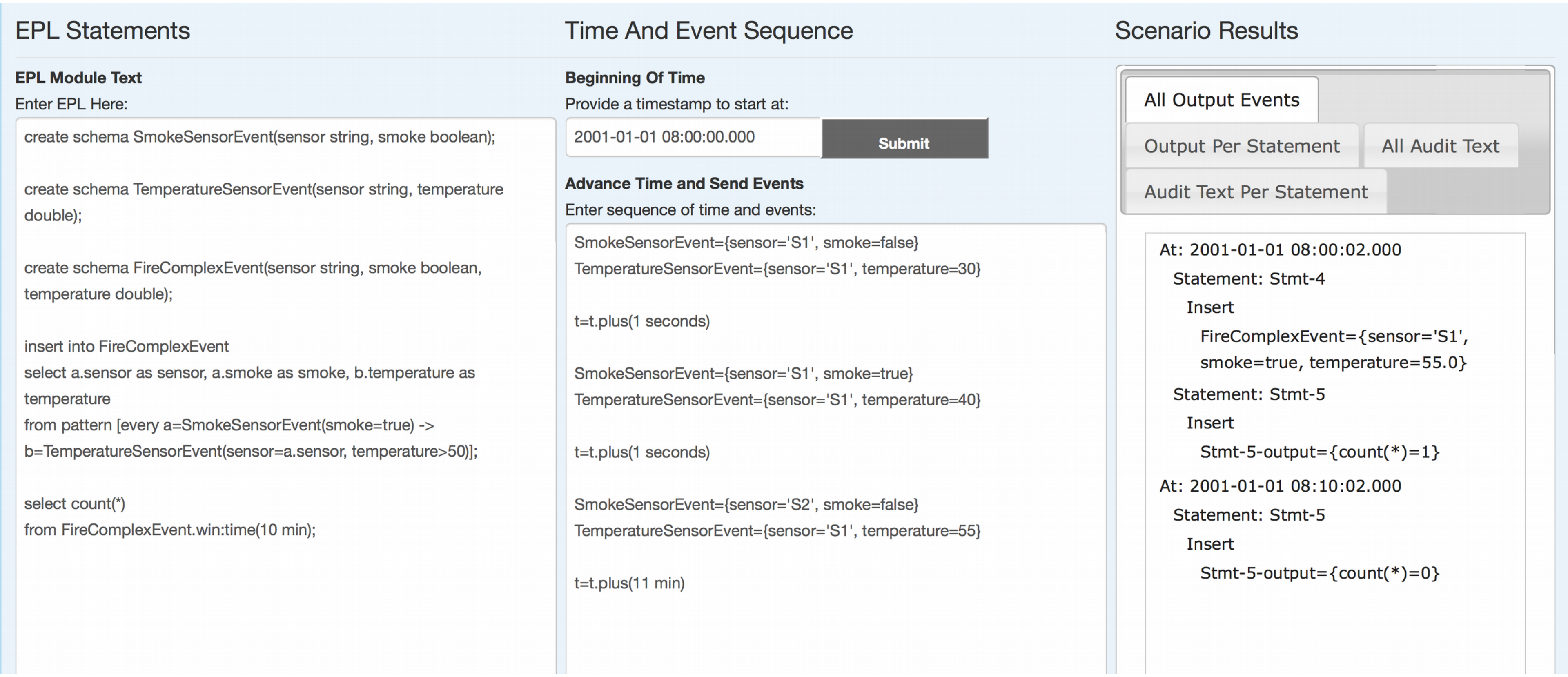
```
select   avg(temperature)
from     TemperatureSensorEvent
```

Running example

```
insert   into FireComplexEvent
select   a.sensor as sensor,
      a.smoke as smoke,
      b.temperature as temperature
from     pattern
      [every a=SmokeSensorEvent(smoke=true)
      ->
      b=TemperatureSensorEvent(
      sensor=a.sensor, temperature>50)];


select   count(*)
from     FireComplexEvent.win:time(10 min);
```

Running example

http://esper-epl-tryout.appspot.com/epltryout/mainform.html



EPL Statements

EPL Module Text

Enter EPL Here:

```
create schema SmokeSensorEvent(sensor string, smoke boolean);

create schema TemperatureSensorEvent(sensor string, temperature
double);

create schema FireComplexEvent(sensor string, smoke boolean,
temperature double);

insert into FireComplexEvent
select a.sensor as sensor, a.smoke as smoke, b.temperature as
temperature
from pattern [every a=SmokeSensorEvent(smoke=true) ->
b=TemperatureSensorEvent(sensor=a.sensor, temperature>50)];

select count(*)
from FireComplexEvent.win:time(10 min);
```

Time And Event Sequence

Beginning Of Time

Provide a timestamp to start at:

2001-01-01 08:00:00.000

Advance Time and Send Events

Enter sequence of time and events:

```
SmokeSensorEvent={sensor='S1', smoke=false}
TemperatureSensorEvent={sensor='S1', temperature=30}

t=t.plus(1 seconds)

SmokeSensorEvent={sensor='S1', smoke=true}
TemperatureSensorEvent={sensor='S1', temperature=40}

t=t.plus(1 seconds)

SmokeSensorEvent={sensor='S2', smoke=false}
TemperatureSensorEvent={sensor='S1', temperature=55}

t=t.plus(11 min)
```

Scenario Results

```
At: 2001-01-01 08:00:02.000
  Statement: Stmt-4
    Insert
      FireComplexEvent={sensor='S1',
      smoke=true, temperature=55.0}
  Statement: Stmt-5
    Insert
      Stmt-5-output={count(*)=1}
At: 2001-01-01 08:10:02.000
  Statement: Stmt-5
    Insert
      Stmt-5-output={count(*)=0}
```

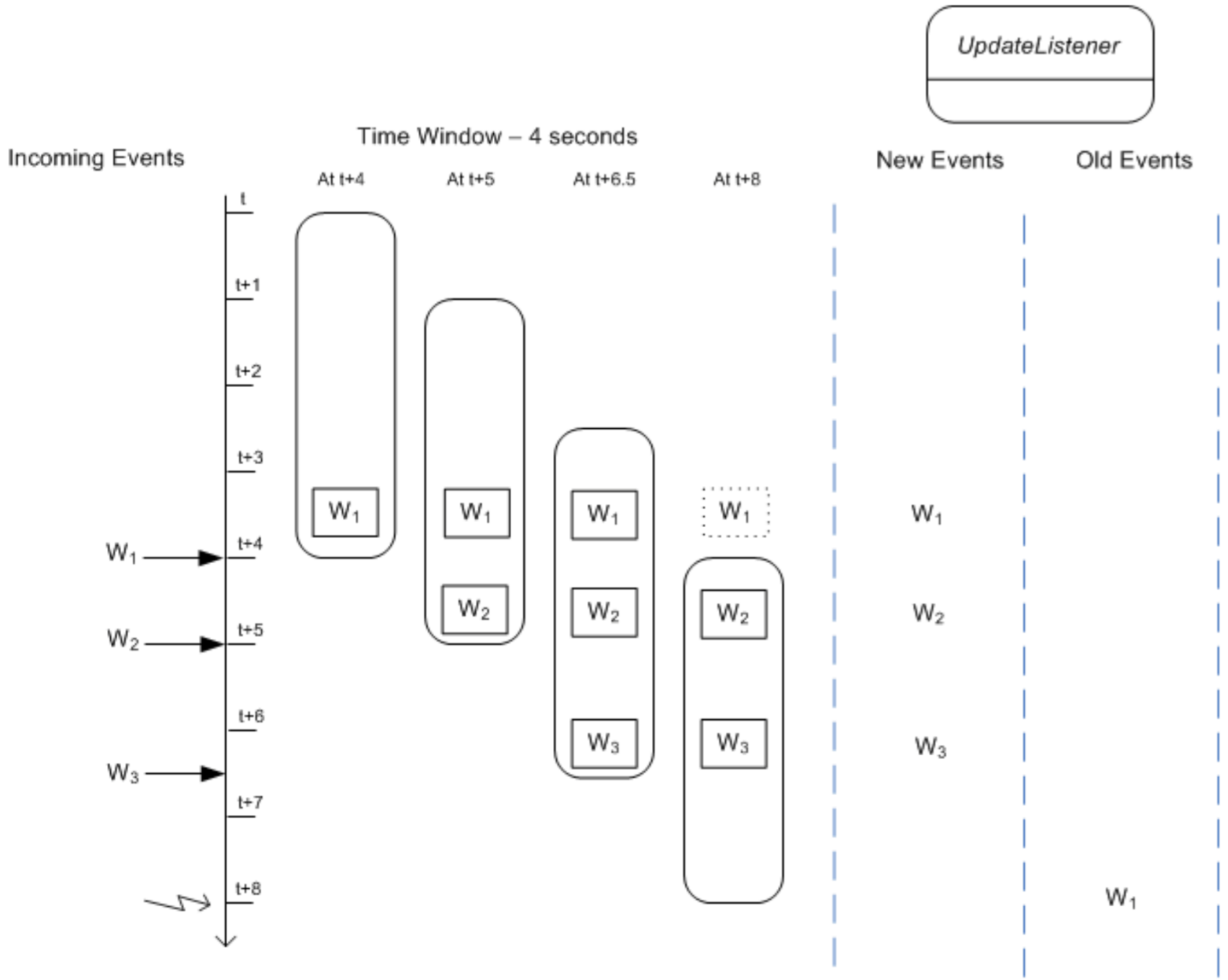
Running example

```
SmokeSensorEvent={sensor='S1', smoke=false}
TemperatureSensorEvent={sensor='S1', temperature=30}
t=t.plus(1 seconds)

SmokeSensorEvent={sensor='S1', smoke=true}
TemperatureSensorEvent={sensor='S1', temperature=40}
t=t.plus(1 seconds)

SmokeSensorEvent={sensor='S2', smoke=false}
TemperatureSensorEvent={sensor='S1', temperature=55}
t=t.plus(11 min)
```

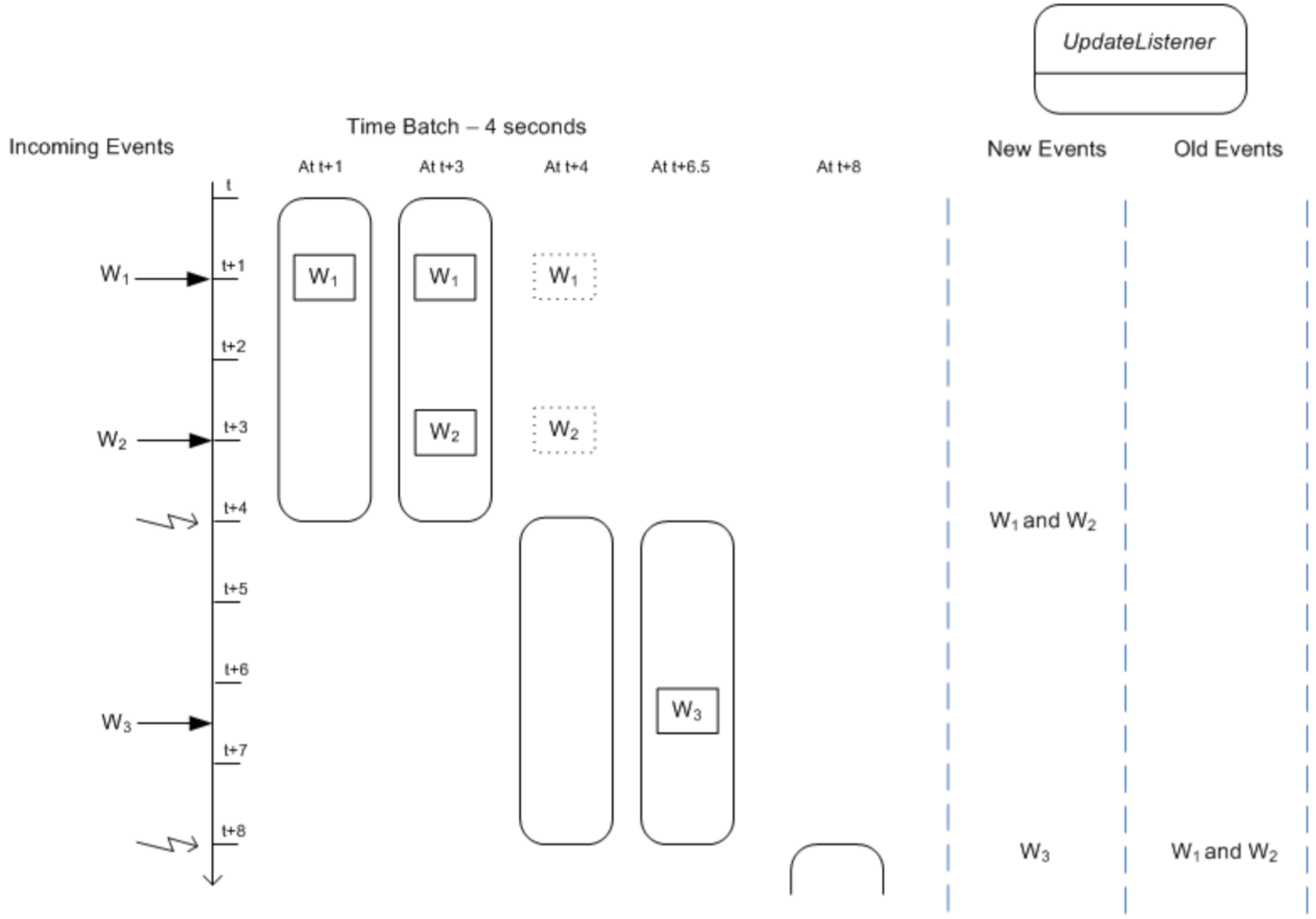
Windows

Type	Syntax	Description
Logical Sliding	win:time(*time_period*)	Sliding window that covers the specified time interval into the past
Logical Tumbling	win:time_batch(*time_period* [, *reference point*] [, *flow control*])	Tumbling window that batches events and releases them every specified time interval, with flow control options
Physical Sliding	win:length(*size*)	Sliding window that covers the specified number of elements into the past
Physical Tumbling	win:length_batch(*size*)	Tumbling window that batches events and releases them when a given minimum number of events has been collected

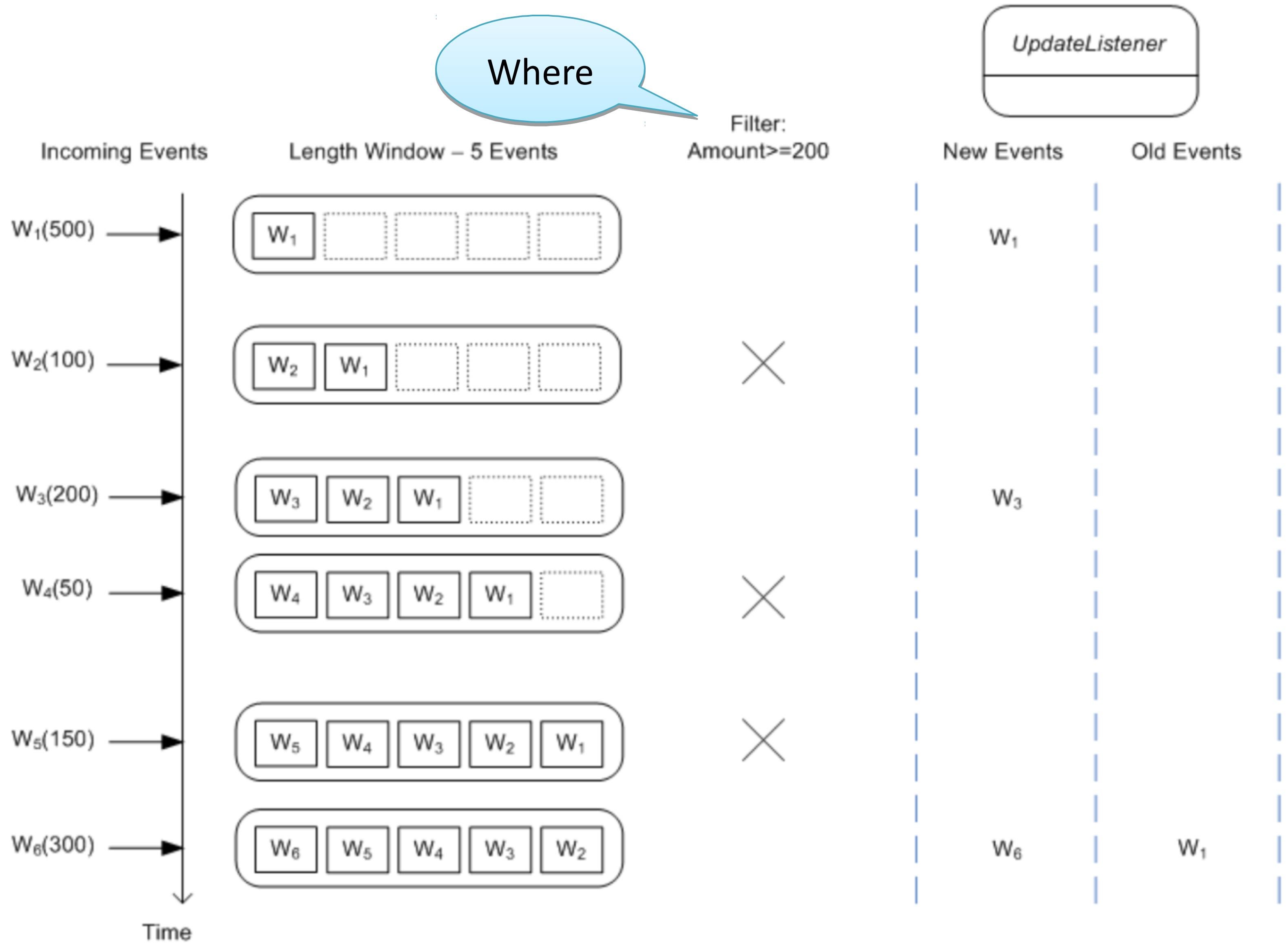
Sliding window



Tumbling window



Physical sliding window



Output control

- The *output* clause is optional in Esper
- It is used to
  - Control the output rate
  - Suppress output events

```
output  [[all | first| last | snapshot]
every     output_rate [seconds | events]]
```

Output control

- Control advancement of sliding windows

```
select   avg(temperature)
from     TemperatureSensorEvent.win:length(4)
output   snapshot every 2 events
```

```
select   avg(temperature)
from     TemperatureSensorEvent.win:time(4 sec)
output   snapshot every 2 sec
```

Pattern matching

- An event pattern emits when one or more event occurrences match the pattern definition
- Patterns can include temporal operators
- Pattern matching is implemented using state machines

Pattern matching

- Content-based event selection

```
TemperatureEventStream(sensor="S0",
temperature>50)
```

- Time-based event observers specify time intervals or time schedules

```
timer:interval(10 seconds)
timer:at(5, *, *, *, *)
```

Fires after 10 seconds

Every 5 minutes
Syntax: minutes, hours, days of month, months, days of week

Pattern matching operators

- Logical operators
  - *and, or, not*
- Temporal operators that operate on event order
  - *-> (followed-by)*
- Creation/termination control
  - *every, every-distinct, [num] and until*
- Guards filter out events and cause termination
  - *timer:within, timer:withinmax* and *while*-expression

Pattern matching

```
select a.sensor from pattern
[every (
   a = SmokeSensorEvent(smoke=true)
   ->
   TemperatureSensorEvent(
      temperature>50,
      sensor=a.sensor)
   where timer:within(2 sec)
)]
```

Pattern matching

- *every expr*
  - When *expr* evaluates to true or false …
  - … the pattern matching for *expr* should re-start

- Without the every operator the pattern matching process does not re-start

Pattern matching

- This pattern fires when encountering an A event and then stops

```
A
```

- This pattern keeps firing when encountering A events, and does not stop

```
every A
```

Pattern matching

A1 B1 B2 A2 A3 B3 A4 B4

`every (A -> B)`

Detect an event A followed by an event B. At the time when B occurs, the pattern matches and restarts looking for the next A event

B1	{A1, B1}
B3	{A2, B3}
B4	{A4, B4}

Pattern matching

A1 B1 B2 A2 A3 B3 A4 B4

`every A -> B`	The pattern fires for every A followed by a B event

B1	{A1, B1}
B3	{A2, B3}, {A3, B3}
B4	{A4, B4}

Pattern matching

A1 B1 B2 A2 A3 B3 A4 B4

```
A -> every B
```

The pattern fires for an A event followed by every B event

B1	{A1, B1}
B2	{A1, B2}
B3	{A1, B3}
B4	{A1, B4}

Pattern matching

A1 B1 B2 A2 A3 B3 A4 B4

`every A -> every B`

The pattern fires for every A event followed by every B event

B1	{A1, B1}
B2	{A1, B2}
B3	{A1, B3}, {A2, B3}, {A3, B3}
B4	{A1, B4}, {A2, B4}, {A3, B4}, {A4, B4}

Pattern matching

- With the every operator
  - Multiple (partial) instances of the same pattern can be active at the same time
  - Each instance can consume some resources when events enter the engine
- End pending instances whenever possible
  - With the *timer:within* construct
  - With the *and not* construct
- Note: the data windows on a pattern do not always limit pattern sub-expression lifetime

Pattern matching

A1 A2 B1

Pattern	Results
every A -> B	{A1, B1}, {A2, B1}
every A -> (B and not A)	{A2, B1}

The *and not* operator causes the sub-expression looking for {A1, B?} to end when A2 arrives

Pattern matching

A1@1　　A2@3　　B1@4

Pattern	Results
every A -> B	{A1, B1}, {A2, B1}
every A -> (B where timer:within(2 sec))	{A2, B1}

The *timer:within* operator causes the sub-expression looking for {A1, B?} to end after 2 seconds

Combine queries

- The insert into clause forwards events to other streams for further downstream processing

```
insert   into FireComplexEvent
select   a.sensor as sensor,
      a.smoke as smoke,
      b.temperature as temperature
from     pattern
      [every a=SmokeSensorEvent(smoke=true)
      ->
      b=TemperatureSensorEvent(
      sensor=a.sensor, temperature>50)];

select   count(*)
from     FireComplexEvent.win:time(10 min);
```

Exercise

- Application scenario: taxi trips in NYC

- Two types of events

```
Pickup(int taxi_id, int location_id)
Dropoff(int taxi_id, int location_id, int amount)
```

- Definitions
  - Route = pair of (pickup location, dropoff location)

Exercise

- Exercise: find the 10 most profitable routes in the last 30 minutes
  - The profitability of a route is the sum of the amounts of all the taxi trips for that route
  - Consider routes that *ended* within the last 30 minutes

Solutions

Assume a stock tick event

`StockTick(String name, int price)`

with the fields name and price representing the name of an company and the associated price for a stock tick.

- Write a query which computes the average prices over the last 30 seconds

```
select avg(price)
from StockTickEvent.win:time(30 sec)
```

Solutions

Assume a stock tick event

`StockTick(String name, int price)`

with the fields name and price representing the name of an company and the associated price for a stock tick.

- Write a query which alerts on each "IBM" stock tick with a price greater then 80 and within the next 60 seconds

```
every StockTickEvent(name="IBM",price>80)
where timer:within(60 seconds)
```

Solutions

Assume a stock tick event

`StockTick(String name, int price)`

with the fields name and price representing the name of an company and the associated price for a stock tick.

- Write a query that returns the average price per name for the last 100 stock ticks

```
select name, avg(price) as averagePrice
from StockTickEvent.win:length(100)
group by name
```

Solutions

- Taxi routes exercise: find the 10 most profitable routes in the last 30 minutes

```
insert into Route
select
pu.pickupLocation as pickupLocation,
do.dropoffLocation as dropoffLocation,
do.amount as amount
from pattern
[every pu=Pickup ->
 (do=Dropoff(taxiId = pu.taxiId)
 where timer:within(30 min))]

select pickupLocation, dropoffLocation, sum(amount) as sum
from Route
group by pickupLocation, dropoffLocation
output all every 1 events
order by sum desc
limit 10
```

MODEL

Why a model?

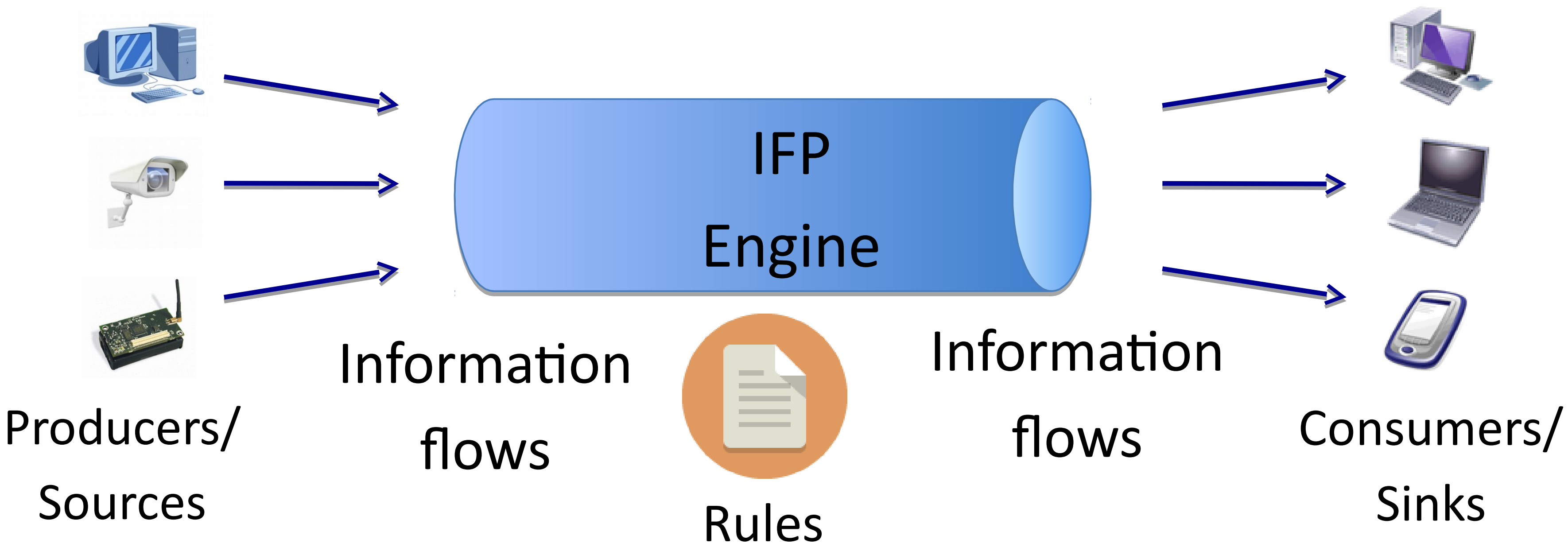
- As discussed in the background
  - Different communities
  - Different vocabularies
  - Different goals
  - Different approaches
  - Different assumptions

Why a model?

- To better understand existing systems
- To classify existing systems
- To help comparing existing systems
- To understand the strengths and the weaknesses of each approach
- To identify solved problems and open issues

Vocabulary

- To avoid biases, we introduce a precise terminology



The IFP domain

- The IFP engine processes incoming flows of information according to a set of processing rules

- The sources produce the input information flows

- The sinks consume the results of processing

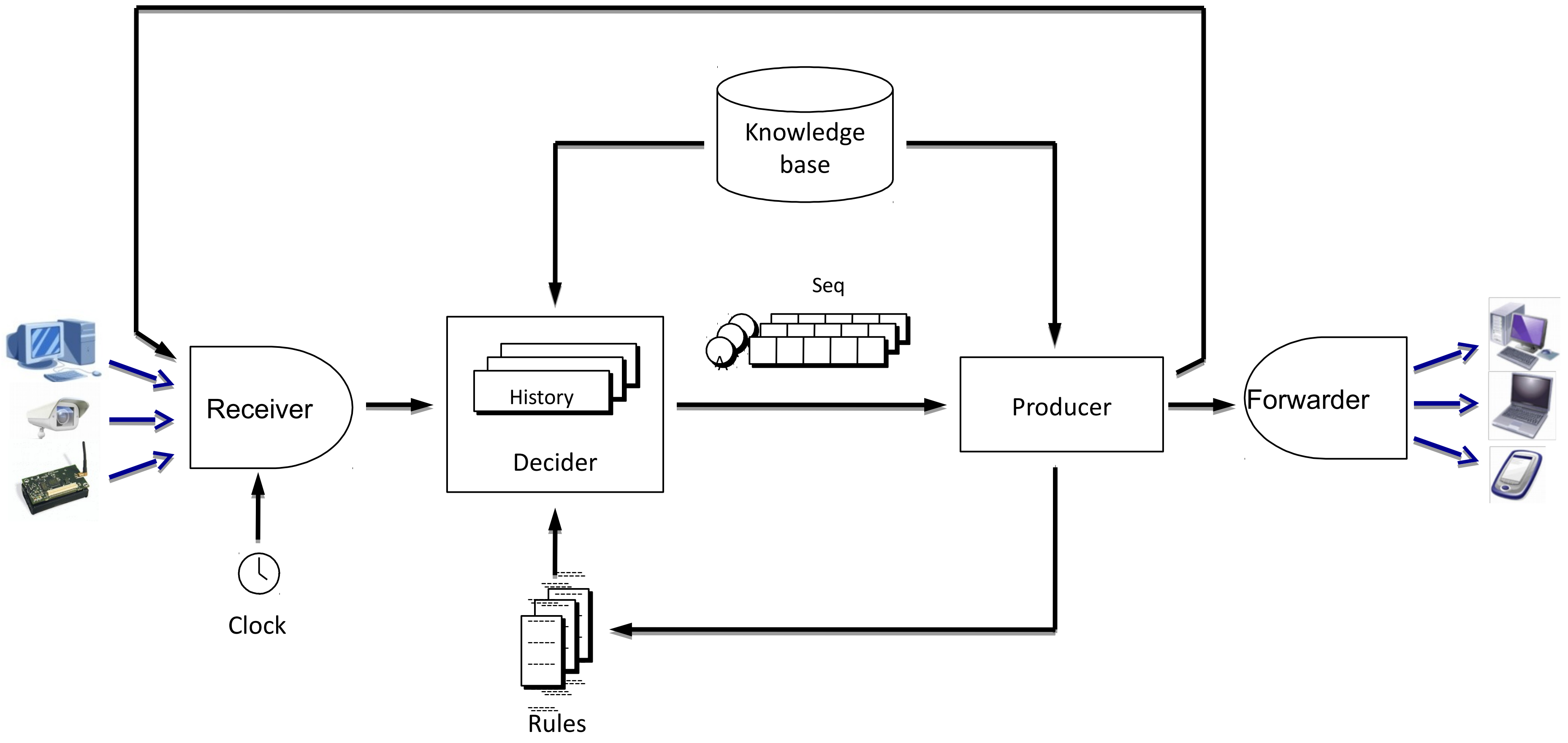
- The rule managers add or remove rules

- Information flows are composed of information items
  - Items part of the same flow are not necessarily ordered nor of the same kind
  - Items part of the same flow are not necessarily of the same kind

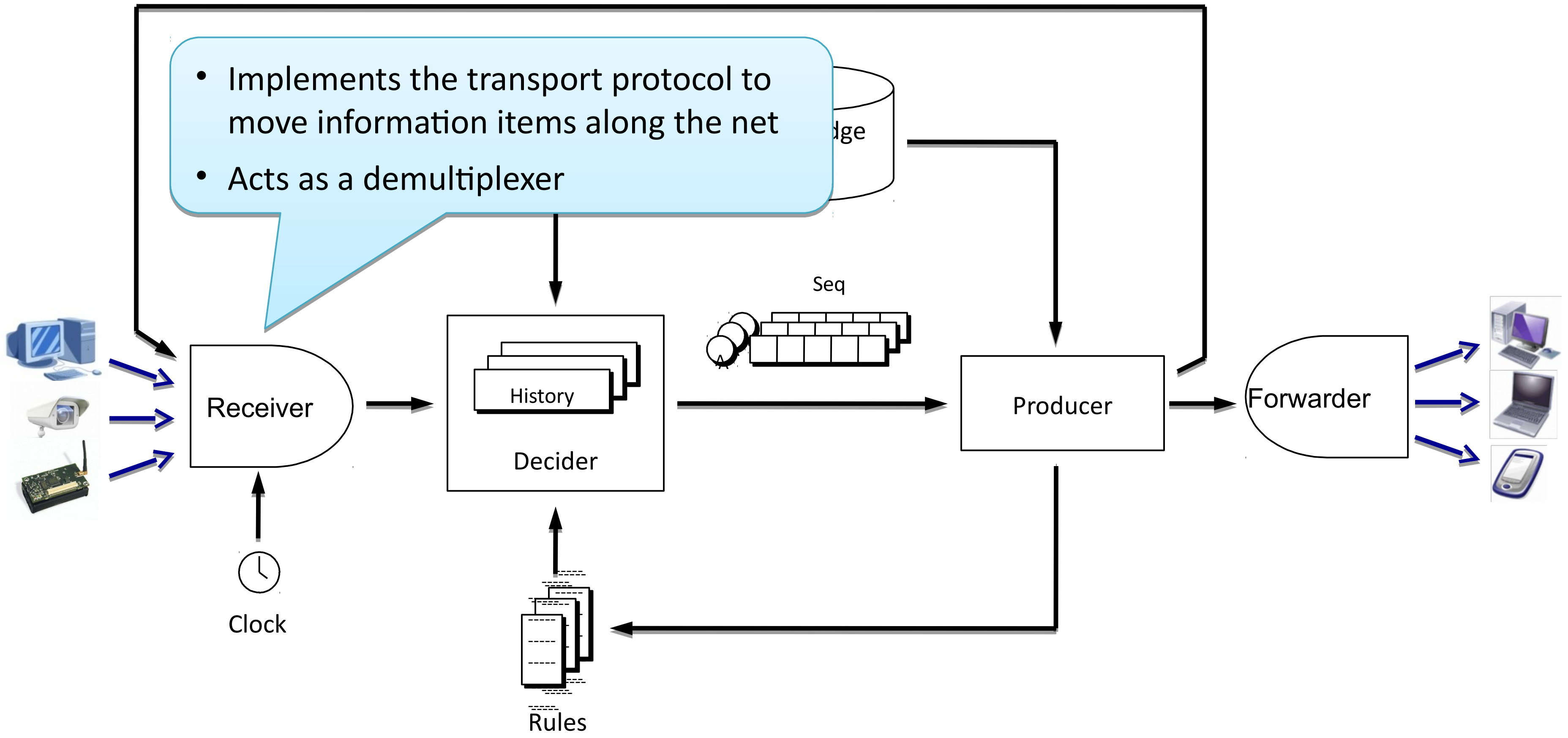
Modeling framework

- Different models to capture different viewpoints
  - Functional model
  - Processing model
  - Deployment model
  - Interaction model
  - Time model
  - Data model
  - Rule model
  - Language model

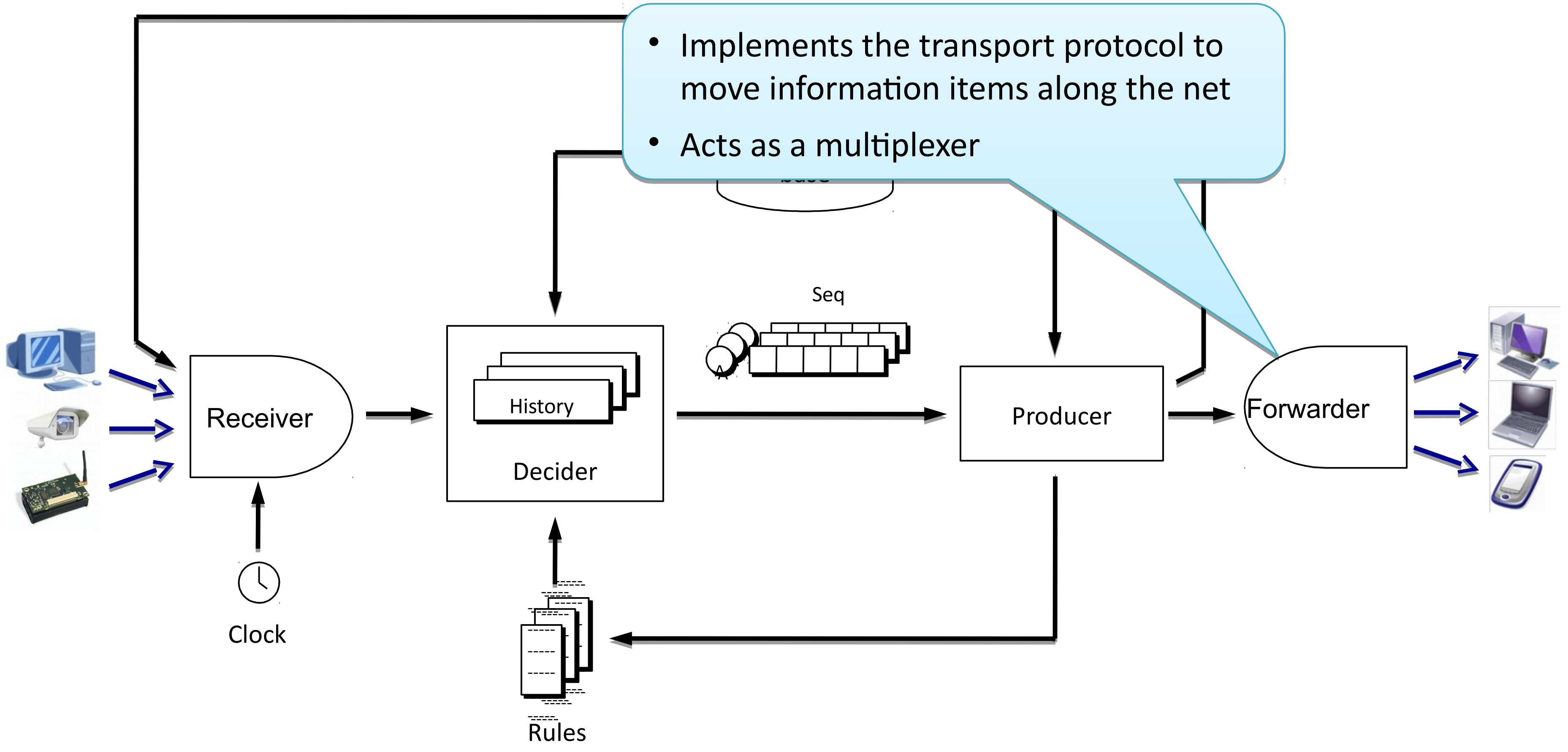
Functional model



Functional model



Functional model



- Implements the transport protocol to move information items along the net
- Acts as a multiplexer

Functional model: assumptions

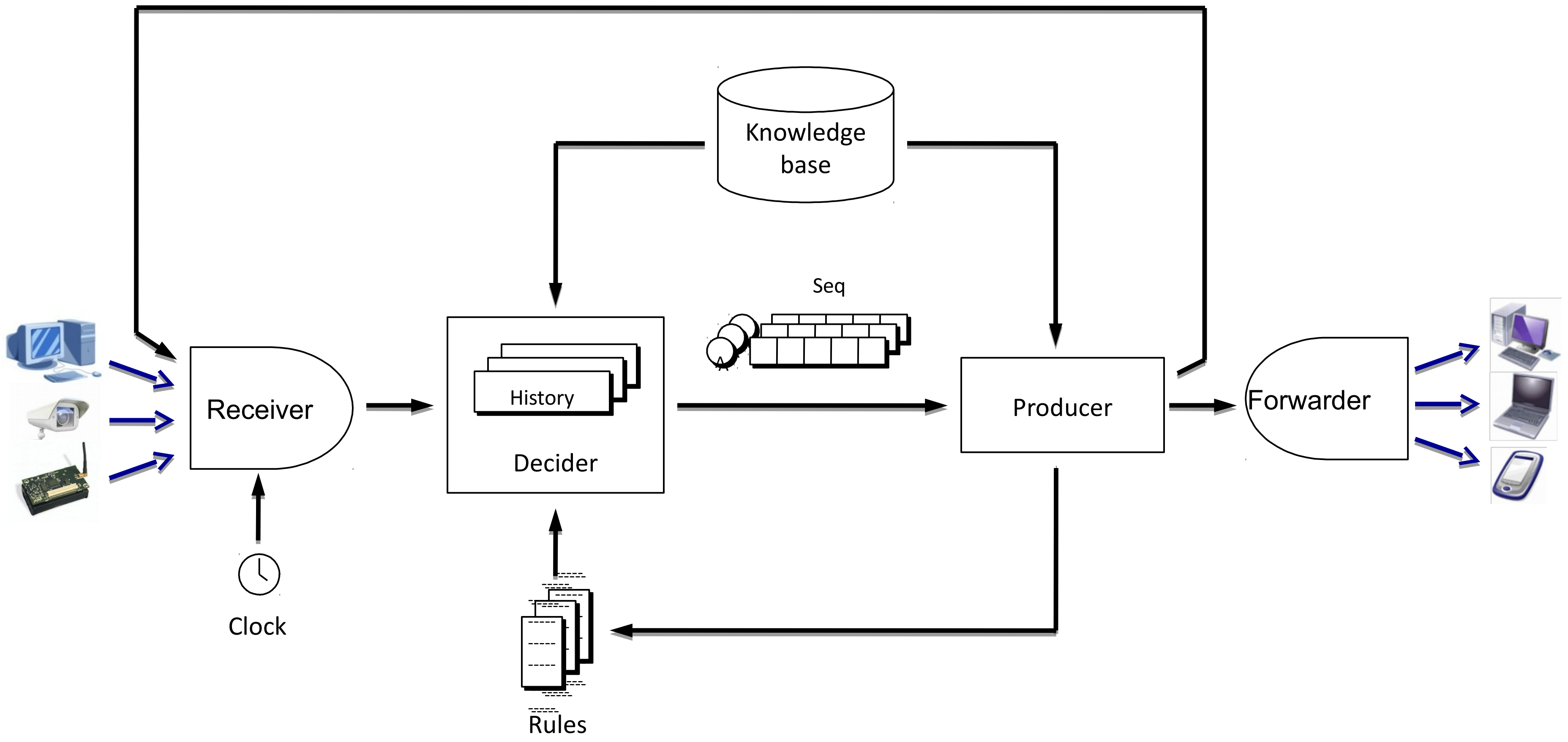
- We assume rules can be (logically) decomposed in two parts: C → A
  - C is the condition
  - A is the action

- Example (in CQL):

```
Select IStream(Count(*))          } action
From F1 [Range 1 Minute]          } condition
Where F1.A > 0
```

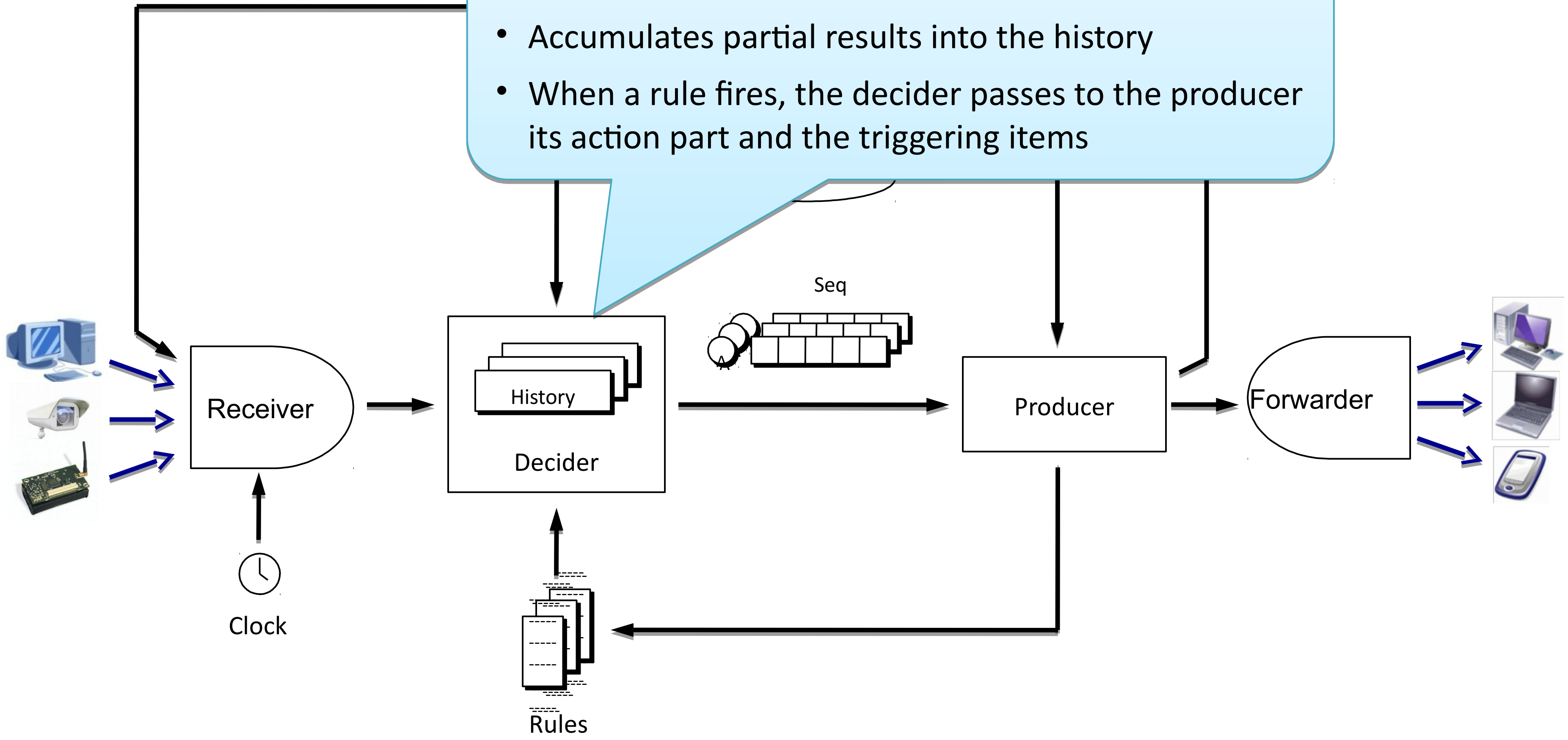
- Accordingly, we split the processing task in two phases
  - The detection phase determines the items that trigger the rule
  - The production phase use those items to produce the output of the rule

Functional model

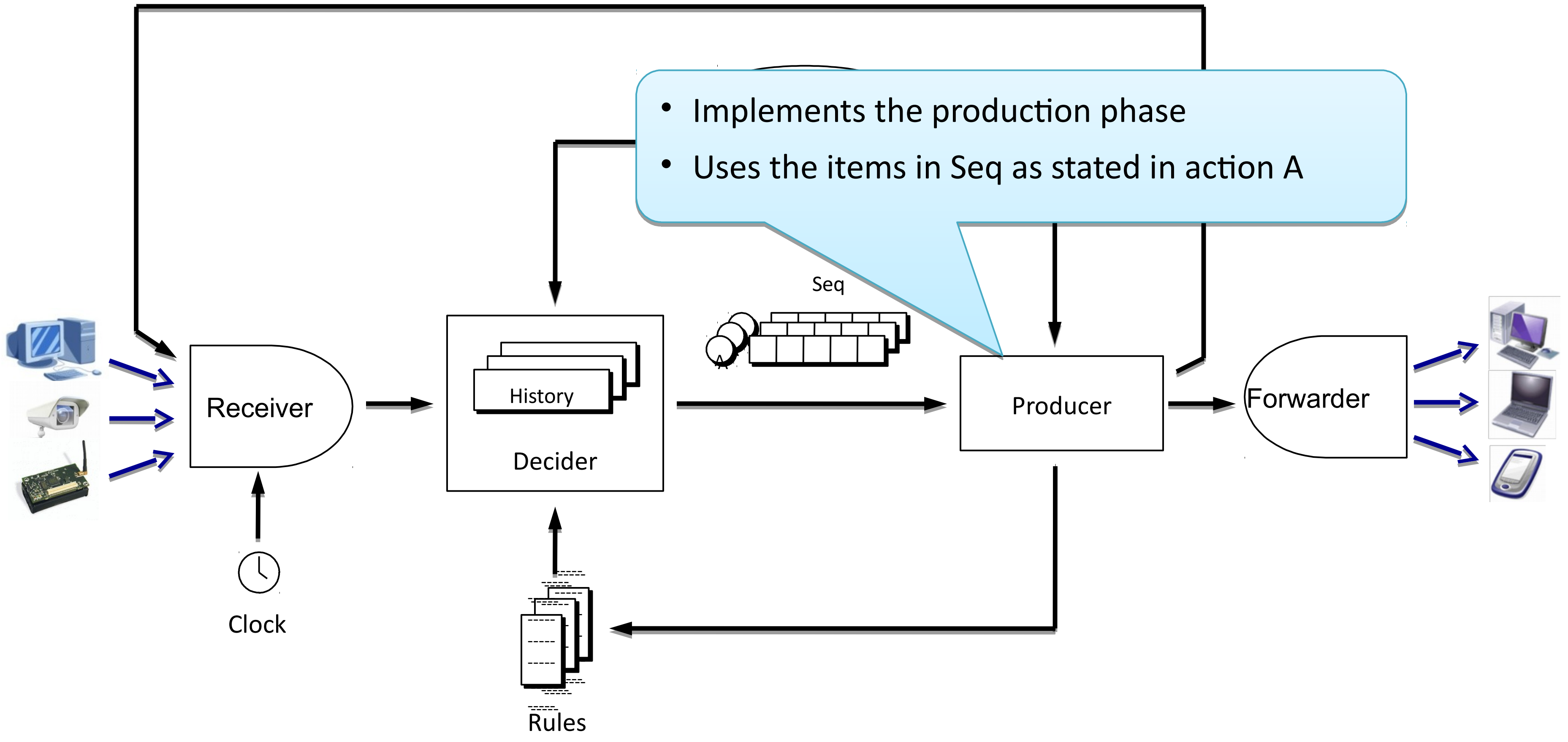


Functional model

- Implements the detection phase
- Accumulates partial results into the history
- When a rule fires, the decider passes to the producer its action part and the triggering items

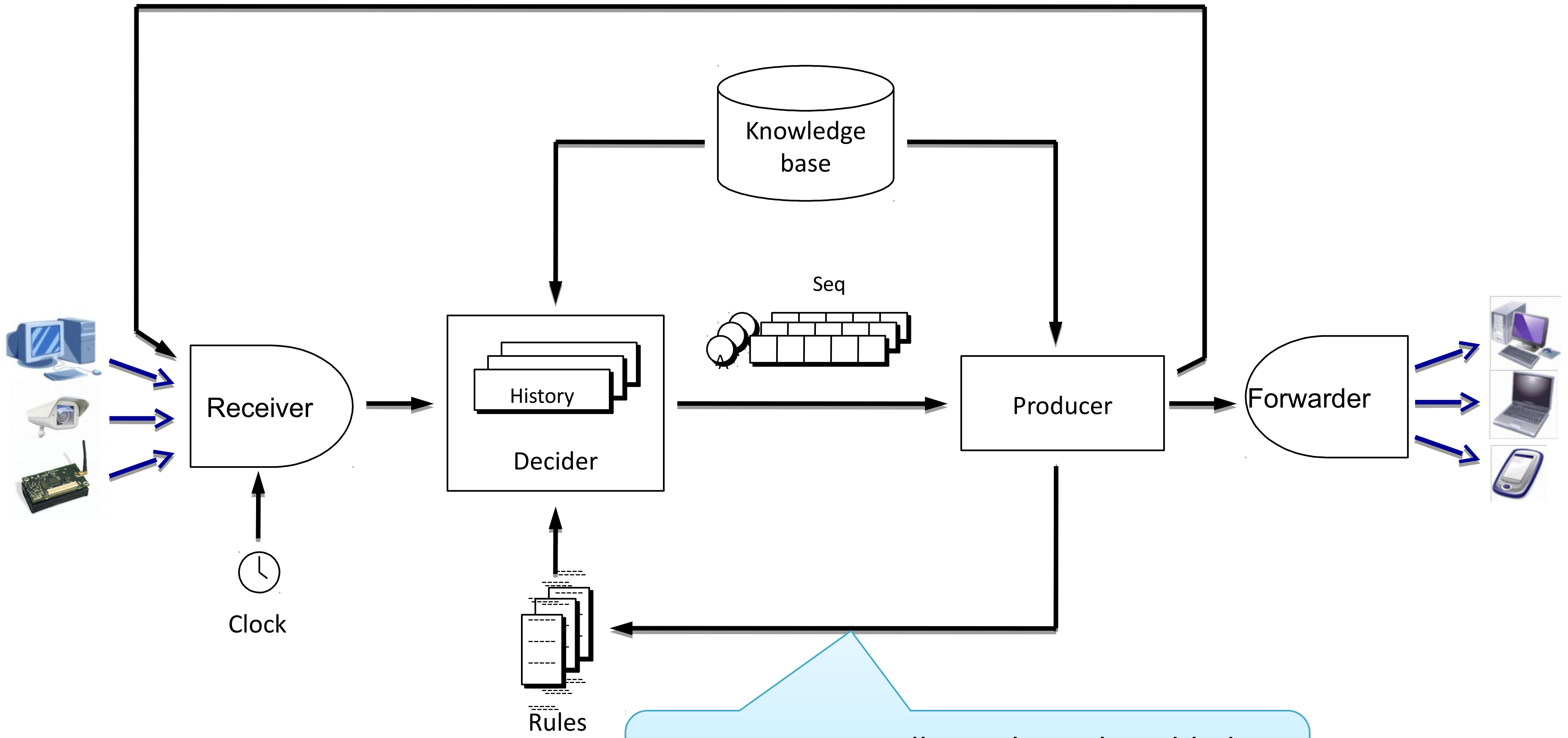


Functional model



- Implements the production phase
- Uses the items in Seq as stated in action A

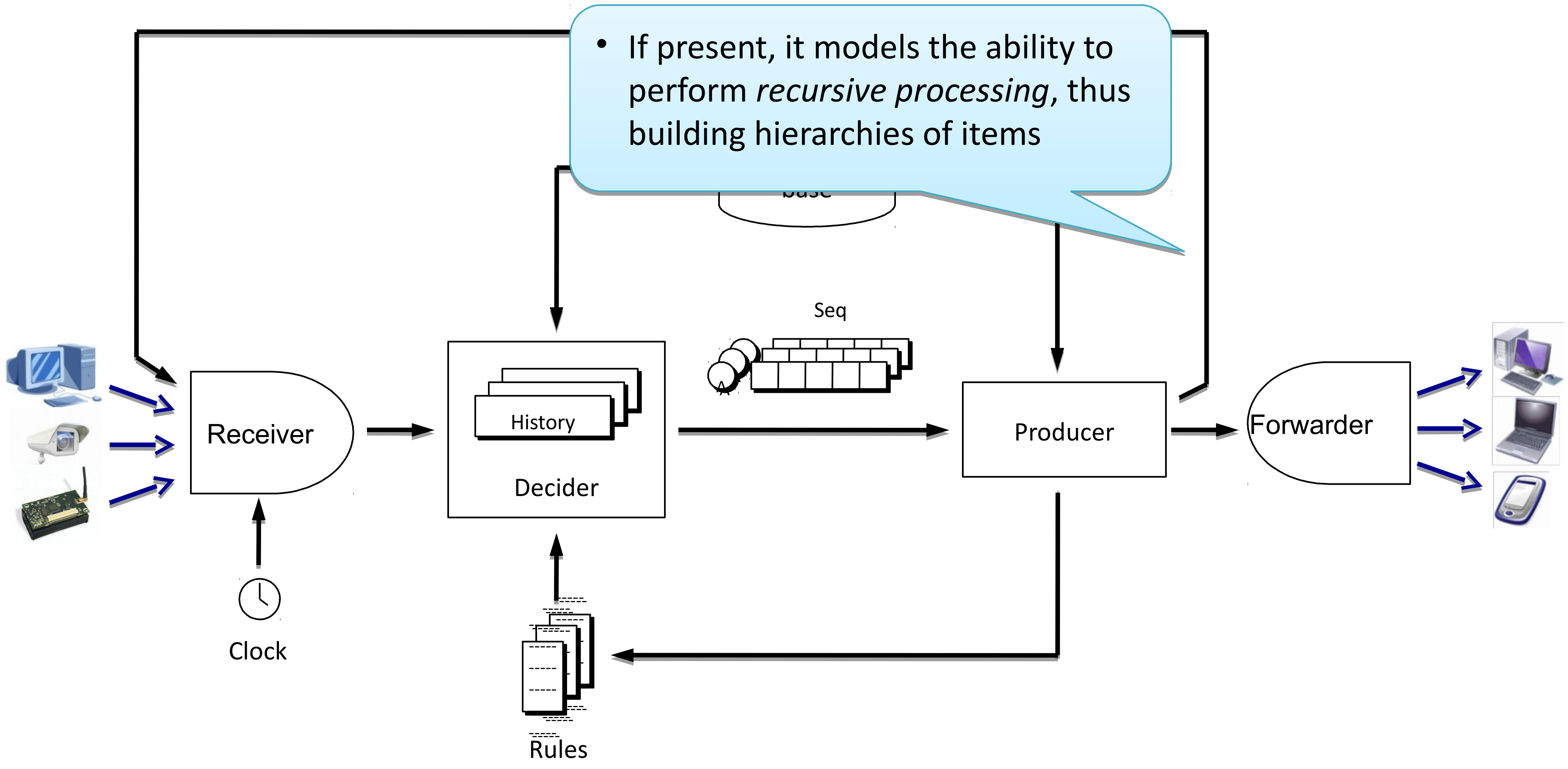
Functional model



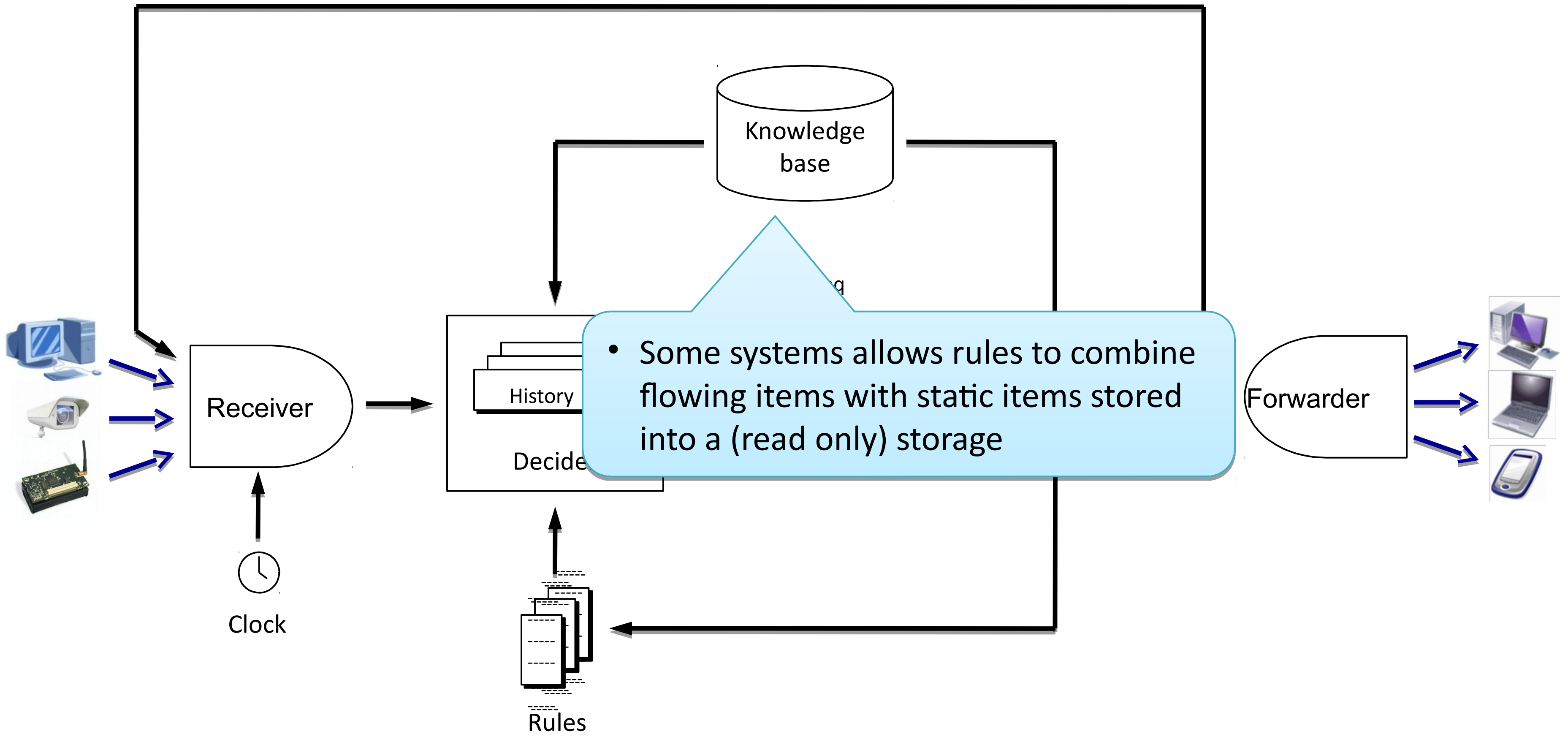
- Some systems allow rules to be added or removed at processing time

Functional model

- If present, it models the ability to perform *recursive processing*, thus building hierarchies of items

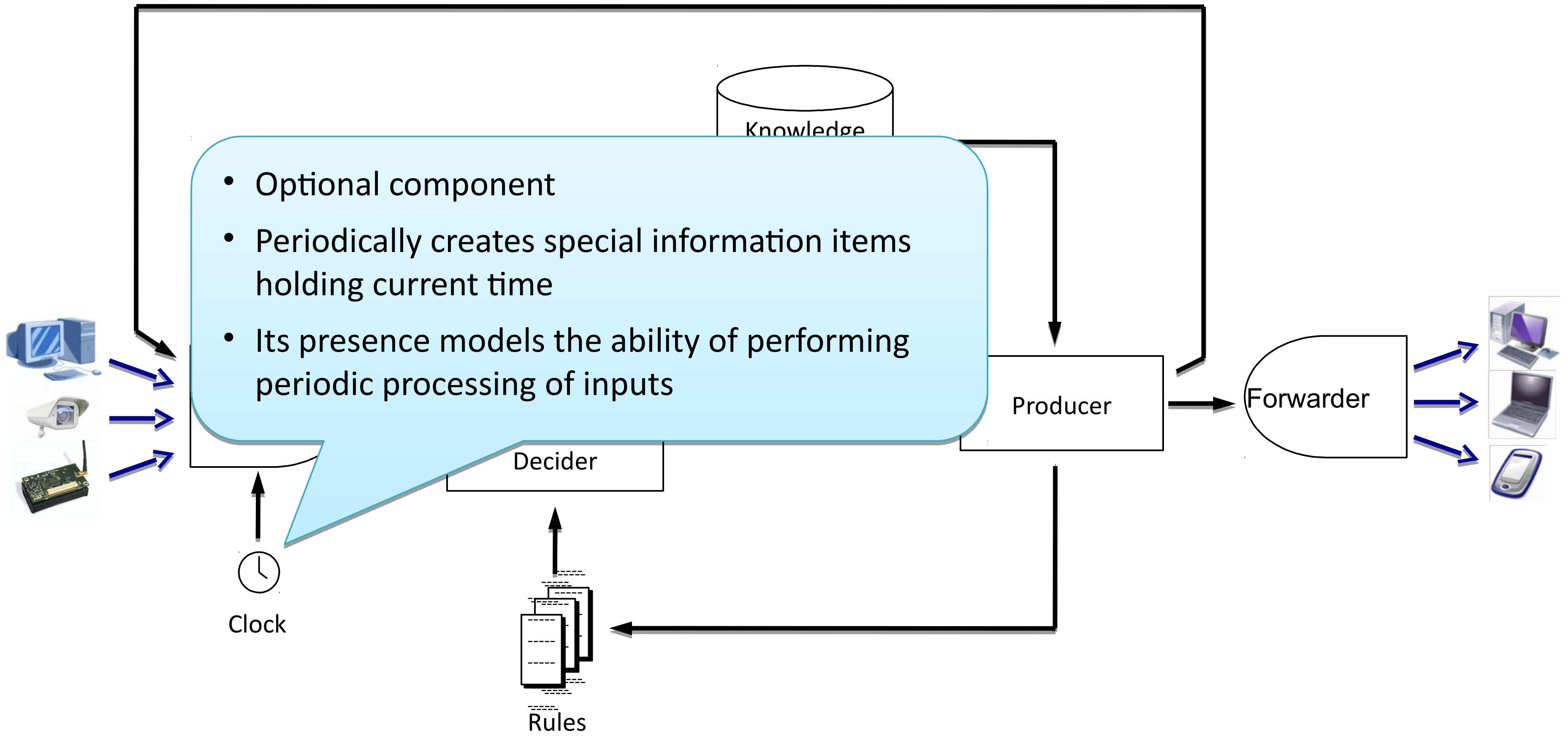


Functional model



- Some systems allows rules to combine flowing items with static items stored into a (read only) storage

Functional model



- Optional component
- Periodically creates special information items holding current time
- Its presence models the ability of performing periodic processing of inputs

Detection-production cycle

- Every new item I entering the engine causes a new detection-production cycle

- If present, the Clock can also generate new items, causing a new cycle

- Each cycle is composed of two phases
  - Detection phase
  - Production phase

Detection phase

- Evaluates all the rules to find those enabled
- Uses the incoming item I, plus the History, plus the data into the Knowledge base, if present
- The item I can be accumulated into the History for partially enabled rules
- The action part of the enabled rules together with the triggering items (A+Seq) is passed to the producer

Production phase

- Produces the output items
- Combining the items that triggered the rule with data present in the Knowledge base, if present
- New items are sent to subscribed sinks (through the Forwarder)…
- …but they could also be sent internally to be processed again (recursive processing)
- In some systems the action part of fired rules may also change the set of deployed rules

Functional model

- Maximum length of Seq a key aspect
  - Bounded: only detection of patterns of fixed length
    - No recursion
    - No time windows
  - Max = 1: decision based only on the current incoming item
    - Stateless operators (filter, project, map, …)
    - Matching in event-based systems

Functional model

- Presence of the Clock models the ability to process rules periodically
  - Available in most "streaming" systems
    - DSMS, RP
  - Not available in many event-based systems
    - CEP

Functional model

- The Knowledge base manages the interaction with static data
  - Available in most DSMS and RP systems
  - Not always available in CEP systems

Functional model

- The presence of a loop from the Producer back to the Receiver models the ability to perform recursive processing
  - Present in several CEP systems
  - DSMS and RP systems sometimes achieve the same expressivity through
    - Nested rules
    - Circular data-flow graph

Functional model

- Support to dynamic rule changes
  - Few systems support it
  - In some cases it can be implemented externally…
  - … through sinks acting also as rule managers

The semantics of processing

- What determines the output of each detection-production cycle?
  - The new item entering the engine
  - The set of deployed rules
  - The items stored into the History
  - The content of the Knowledge Base

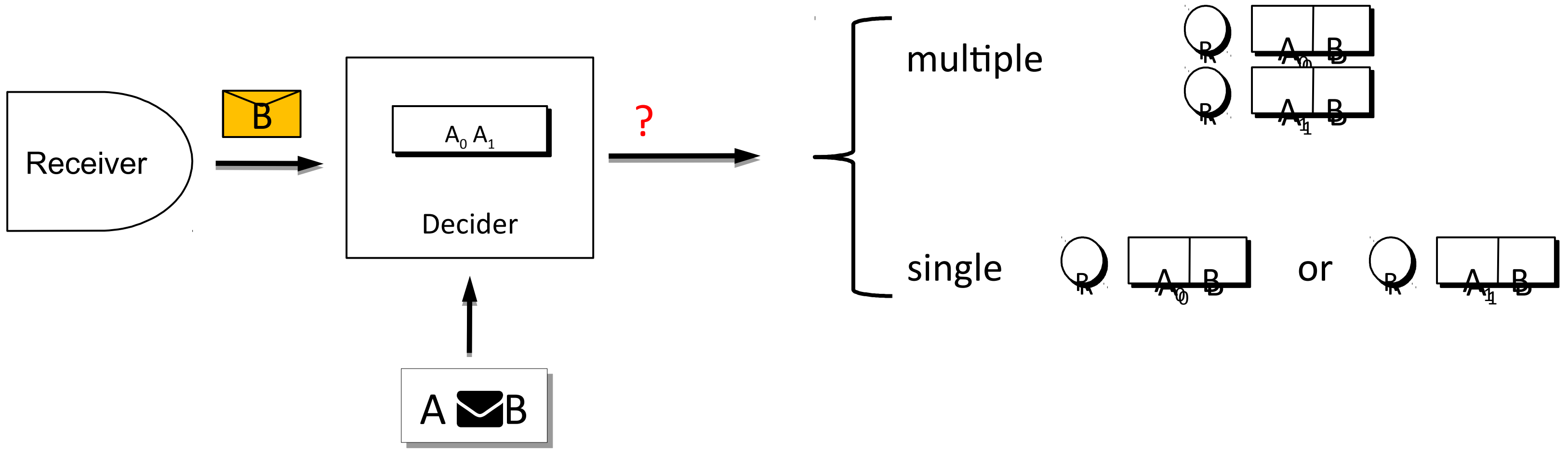
- Is this enough?

Processing model

- Three policies affect the behavior of the system
  - The selection policy
  - The consumption policy
  - The load shedding policy

Selection policy

- Determines whether a rule fires once or multiple times
  - Also determines which items are selected from the History



Selection policy

- Most systems adopt a multiple selection policy

- Is it adequate? Not always …
  - Example rule: Alert fire when smoke and high temperature are detected in a short time frame
  - 10 sensors read high temperature
  - Immediately after one sensor detects smoke
  - One would like to receive a single alert, not 10

- A few systems allow this policy to be programmed…
  - … some of them on a per-rule base
    - E.g., Esper's *every* operator

Selection policy: the TESLA case

- TESLA (language of the T-Rex CEP system) provides a customizable selection policy on a per rule base
  - Example: Multiple selection

```
define   Fire(area:
from     Smoke(area
         each Temp(                                . from Smoke
where    area=Smoke                                 ue
```

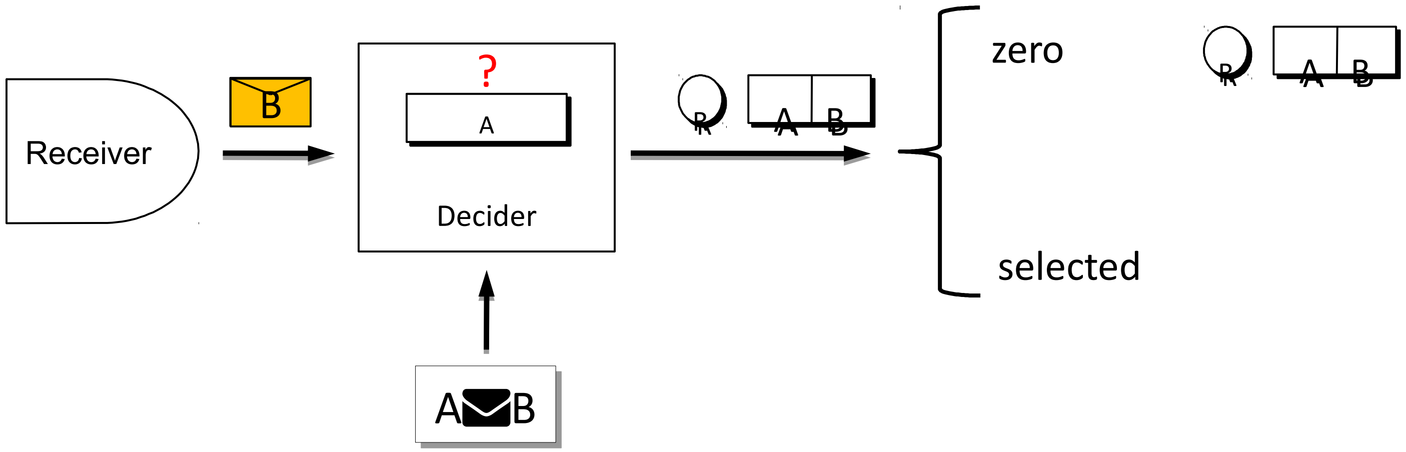
- TESLA also offers:
  - first … within
  - n-first … within   n-last … within

  - Example: Single selecti

```
define   Fire(area: string, measuredTemp: double)
from     Smoke(area=$a) and
         last Temp(area=$a and val>50) within 1min. from Smoke
where    area=Smoke.area and measuredTemp=Temp.val
```

Consumption policy

- Determines how the history changes after firing of a rule / what happens when new items enter the Decider



Consumption policy: considerations

- Most systems couple a multiple selection policy with a zero consumption policy
  - This is the common case with DSMSs, which use (sliding) windows to select relevant events

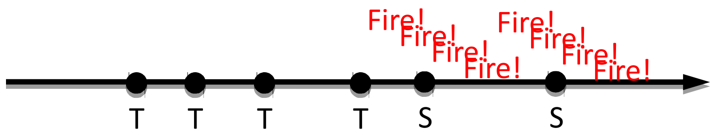
```
Select  IStream(Smoke.area)
From  Smoke[Range 1 min], Temp[Range 1 min]
Where Smoke.area = Temp.area AND Temp.val > 50
```

- The systems that offer a programmable selection policy, often offer a programmable consumption policy, too

Consumption policy: The TESLA case

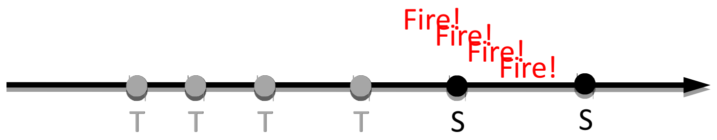
- Zero consumption policy

```
define   Fire(area: string, measuredTemp: double)
from     Smoke(area=$a) and
         each Temp(area=$a and val>50)
         within 1min. from Smoke
where    area=Smoke.area and measuredTemp=Temp.value
```



- Selected consumption policy

```
define    Fire(area: string, measuredTemp: double)
from      Smoke(area=$a) and
          each Temp(area=$a and val>50)
          within 1min. from Smoke
where     area=Smoke.area and measuredTemp=Temp.value
consuming Temp
```



Load shedding policy

- Defines how to manage bursts of input data

- Accumulate pending items in the Receiver
  - Side effect: the delay increases
- Discard some items
  - Side effect: the results might be incomplete

Load shedding policy

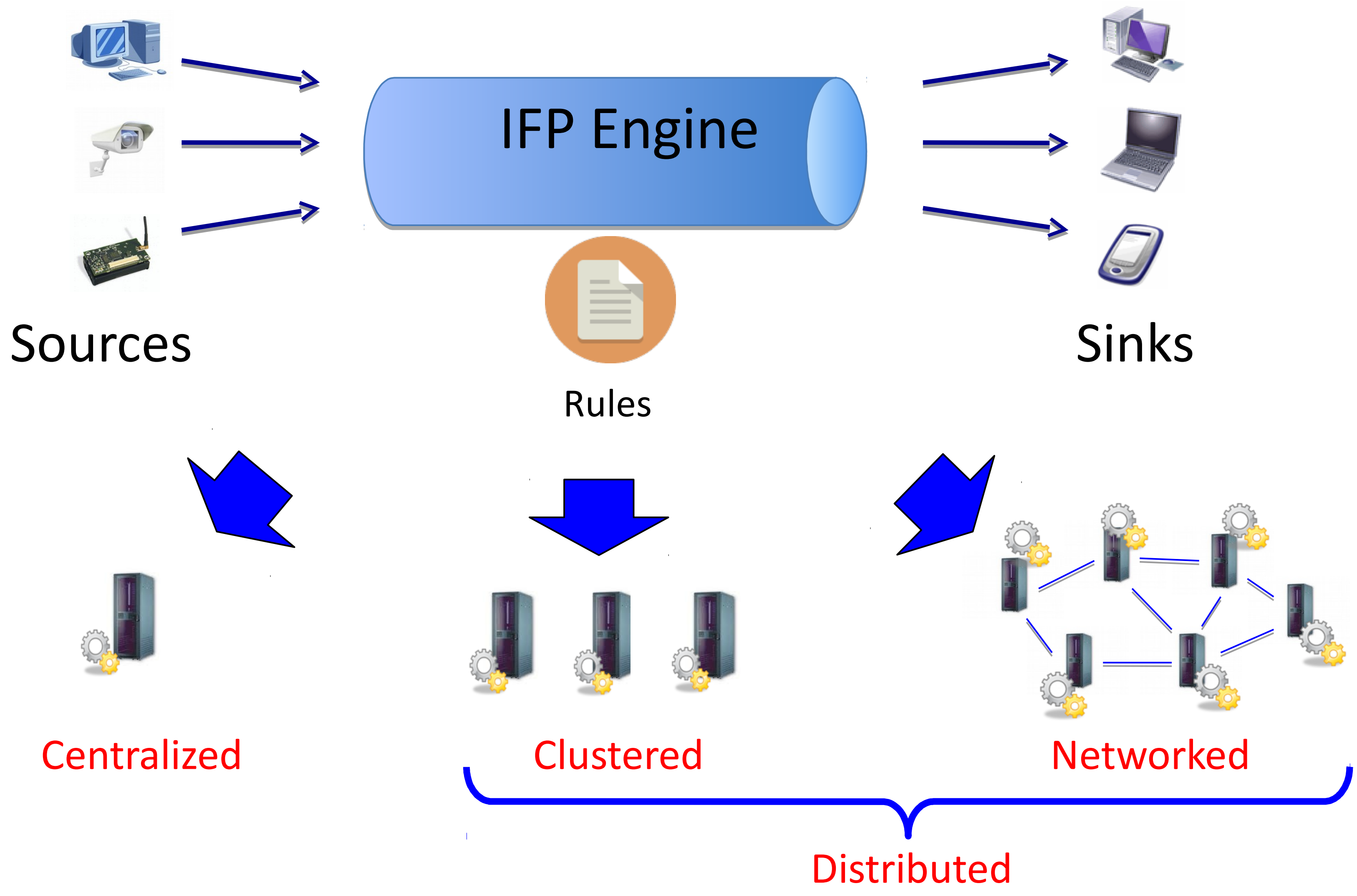
- It may seem a system issue …
  - To be solved by the Receiver
- … but it strongly impacts the results produced
  - The semantics of the rules

- Some systems enable rule managers to specify load shedding policies on a per-rule basis
  - For instance, the Aurora DSMS allows rules to specify QoS requirements and sheds input to stay within the specified limits with the available resource

Deployment model

- IFP applications may include a large number of sources and sinks
  - Possibly dispersed over a wide geographical area
- It becomes important to consider the *deployment architecture* of the engine
  - How the components of the functional model can be distributed to achieve scalability

Deployment model



Deployment model

Clustered

- Processing nodes are geographically co-located
- Large bandwidth
- Limited communication delay
- Potentially adopting shared memory model

Networked

- Processing nodes are geographically distributed
- Bandwidth can be a bottleneck
- Communication delay can be relatively high
- No shared memory

Deployment model

- Many systems adopt a centralized solution
- Some systems have been explicitly designed for cluster deployments
- Only few systems target networked deployments
  - In most cases, deployment/configuration is not automatic

Distribution: why?

- More processing power to reduce processing latency
  - Current algorithms already very efficient but …
  - … certain computations may still introduce bottlenecks
    - Complex aggregations
    - Large volumes of streaming data
    - Large volumes of background data

- Independent operations can be carried out in parallel on multiple machines

Distribution: why?

- Scalability in the number of rules
  - Different rules on different machines
- Scalability in the number of sources and sinks
  - Input and output connections
  - One machine can (efficiently) support only a limited number of open connections

Distribution: why?

- The application scenario is *intrinsically* distributed
  - Distributed sources, sinks, background knowledge
  - Network can become the bottleneck
    - Bandwidth
    - Delay
  - Need to consider how to route and where to process your information
    - E.g., high frequency traders locate their machine close to the sources

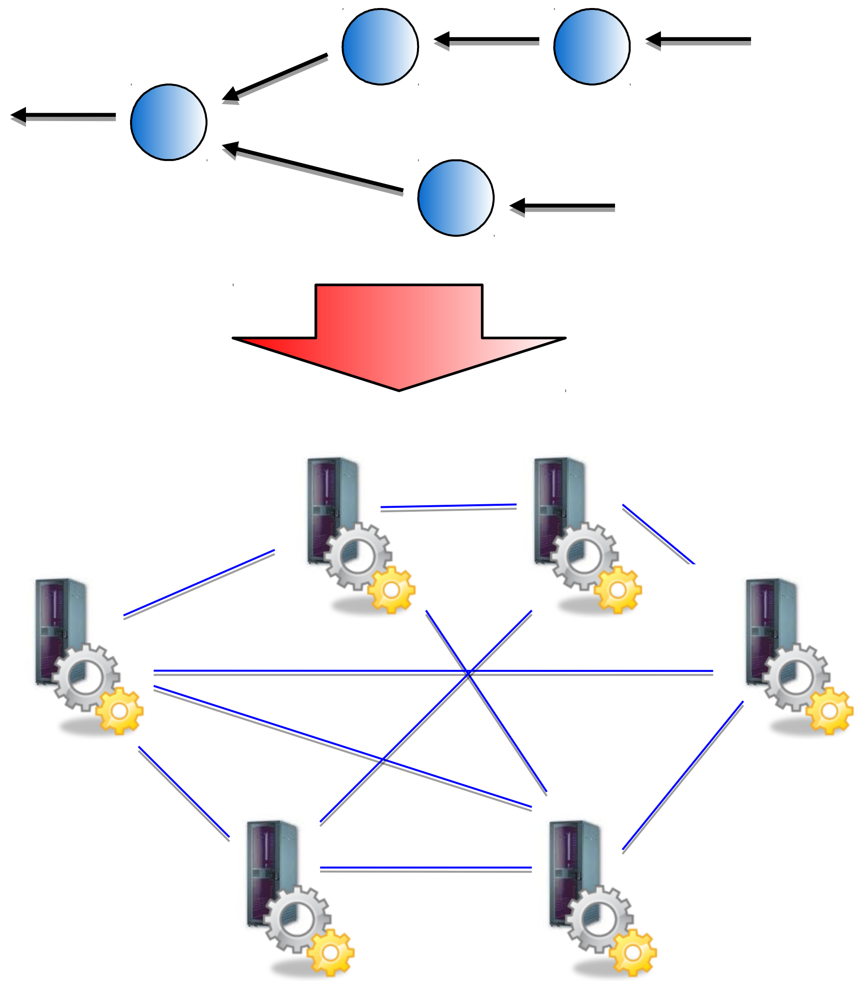
Distribution: why?

- Resource-constrained nodes
  - Sensors
  - Mobile devices

- Offload (only part of) the computation!
  - Perform part of the computation in the mobile source to reduce network communication (battery!)

Deployment model

- Automatic distribution of processing introduces the *operator placement* problem
- Given a set of rules (composed of operators) and a set of nodes
  - How to split the processing load
  - How to assign operators to available nodes
- In other words
  - Given a *processing network*
  - How to map it onto the physical network of nodes



Operator placement

- The operator placement problem is still open
  - Several proposals
  - Different goals
  - Difficult to compare solutions and results
    - Even in its simplest form the problem is NP-hard

Operator placement: goals

- Load
  - Aggregate CPU usage of all the operators deployed in each node
  - Different variants
    - Minimize average load
    - Minimize maximum load (avoid/limit bottlenecks)
    - Minimize load variance (avoid/limit bursts)

Operator placement: goals

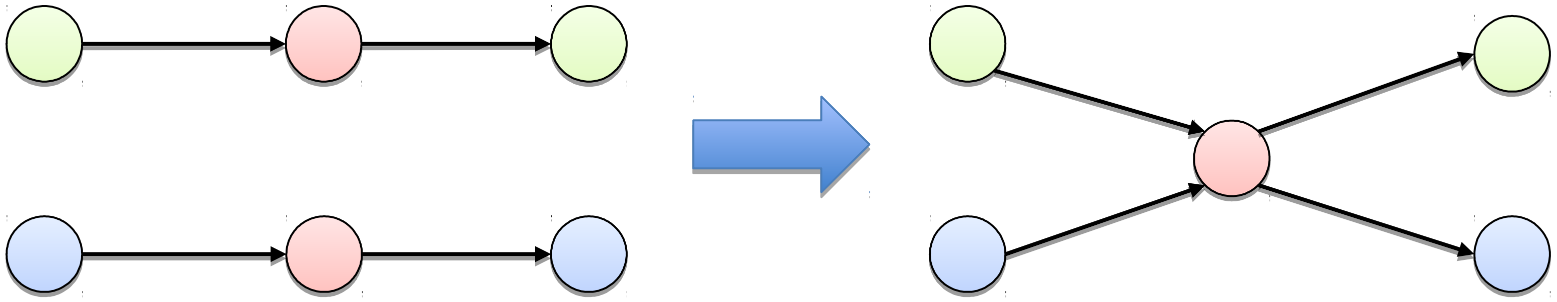
- Latency and load
  - Initial placement
    - Based on network cost (latency)
  - Load-balancing strategy
    - To adapt to changes in data and resource conditions

Operator placement: goals

- Latency and bandwidth
  - Minimize network usage $u = \sum DR(L)*Lat(L)$
    - DR(L) data rate over link L
    - Lat(L) latency (cost) of link L
  - Tolerate paths with additional latency …
  - … if they reduce the overall stream bandwidth

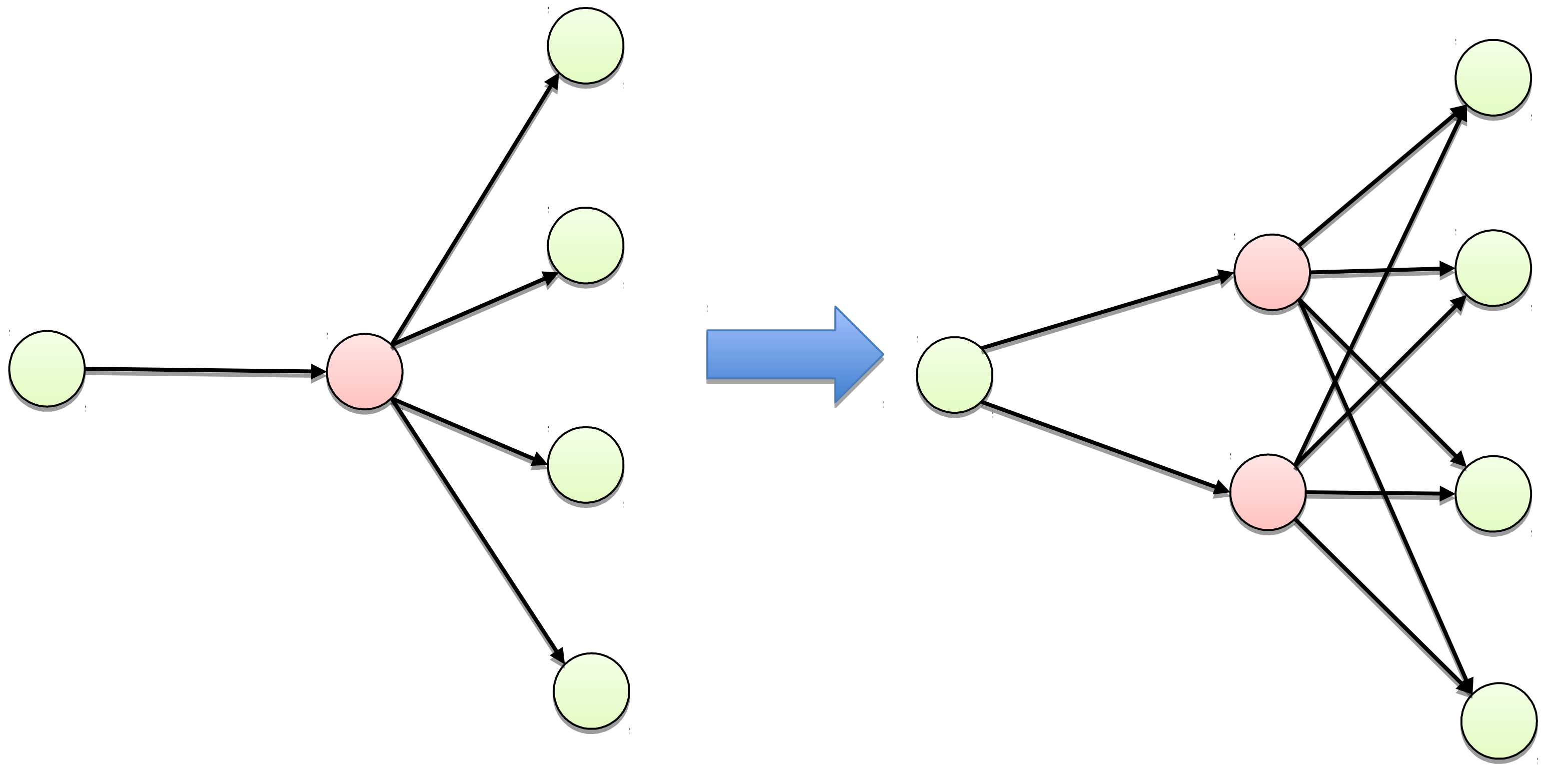
Operator placement

- Optimizations: operator reuse

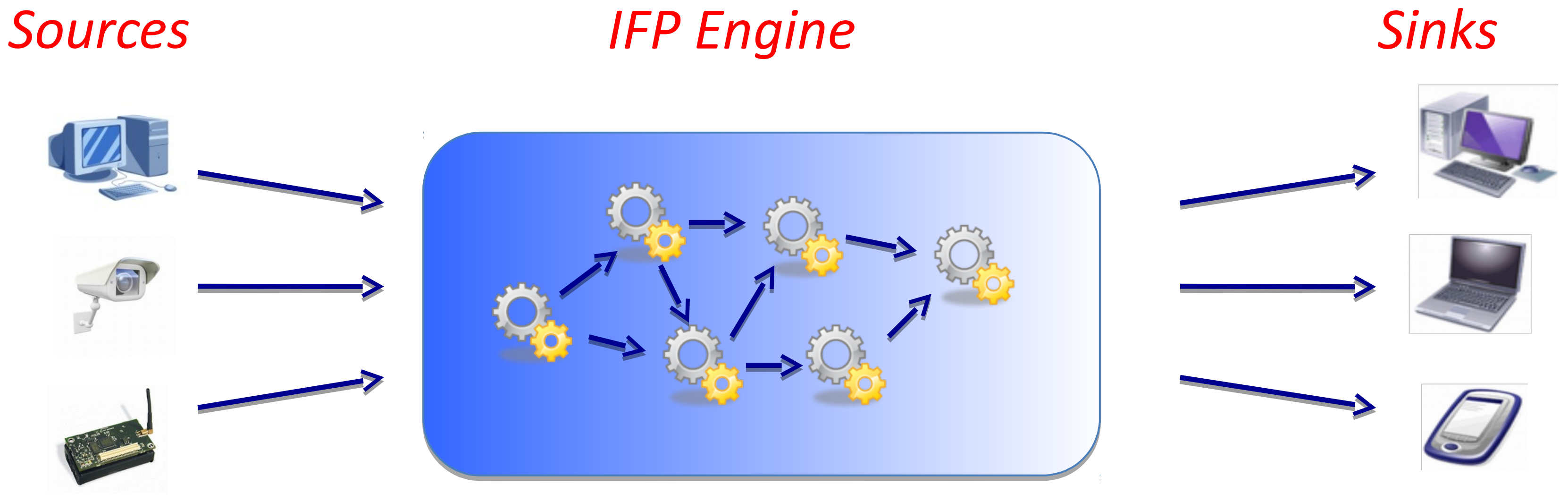


Operator placement

- Optimizations: operator replication

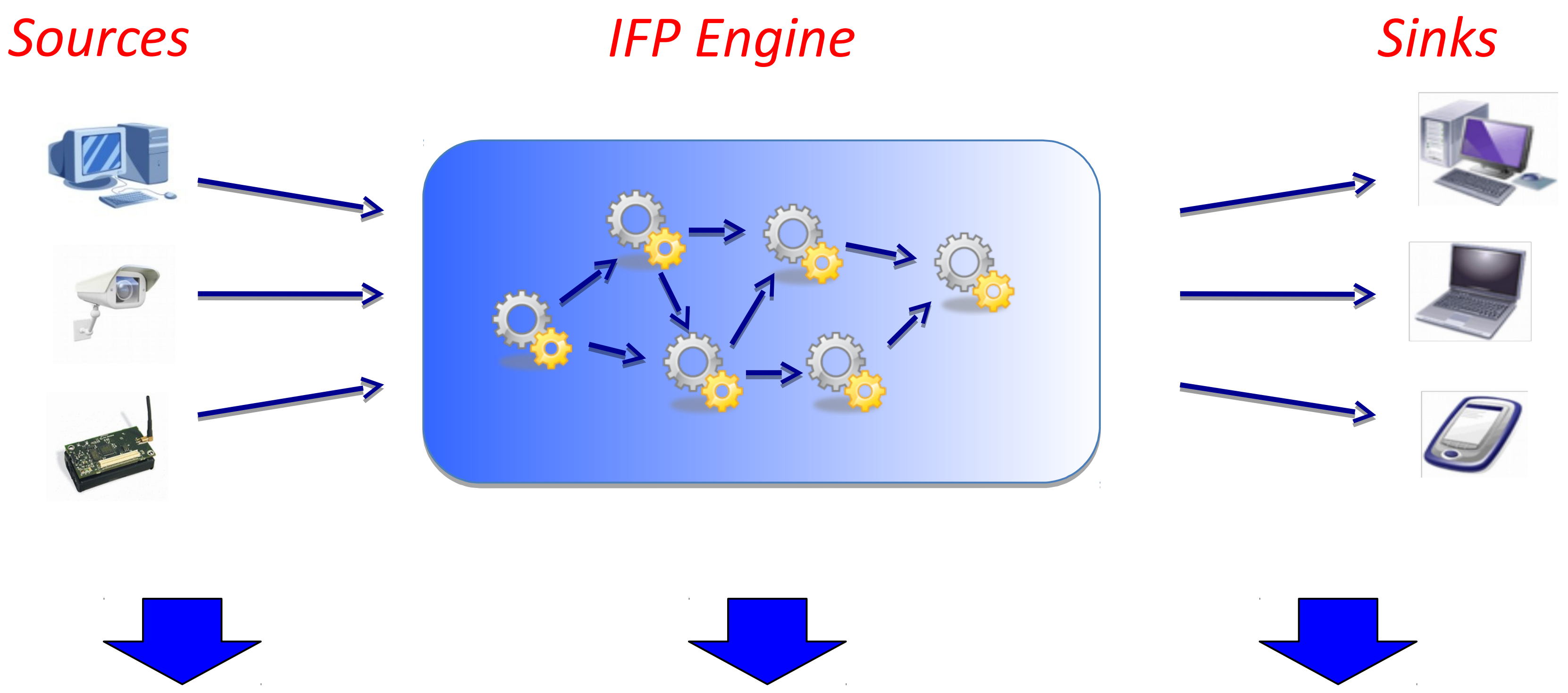


Interaction model



- Models the interactions between components in an IFP systems
  - Who starts the communication?

Interaction model



Observation Model

- Push
- Pull

Forwarding Model

- Push
- Pull

Notification Model

- Push
- Pull

Time model

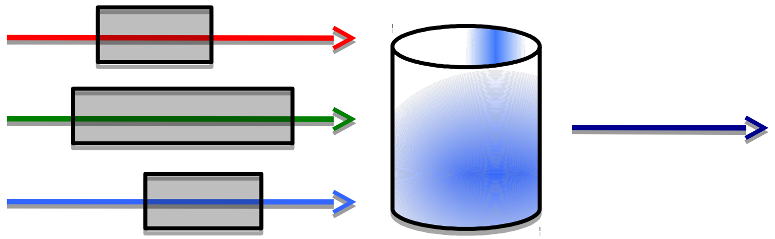
- Relationship between information items and passing of time
- Ability of an IFP system to associate some kind of ordering relationship to information items
- We identified 4 classes:
    1. Stream-only
    2. Causal
    3. Absolute
    4. Interval

Stream-only time model

- Used in DSMS / RP
- Timestamps may be present or not
- When present, they are used only to order items before entering the engine, then they are forgotten
- They are not exposed to the language
  - With the exception of windows
- Ordering in output streams is conceptually separate from the ordering in input streams

<u>CQL/Stream</u>

```
Select DStream(*)
From   F1[Rows 5],
       F2[Range 1 Minute]
Where  F1.A = F2.A
```

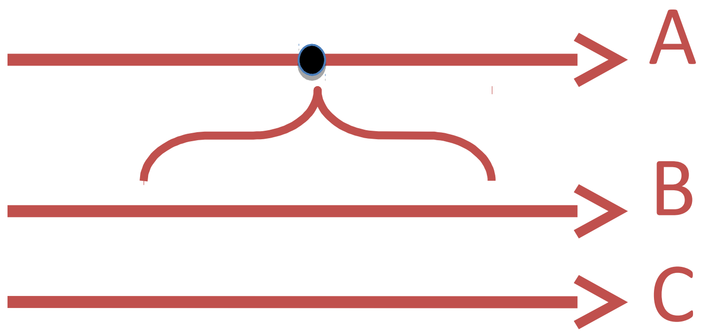


Causal time model

- Each item has a label reflecting some kind of causal relationship
- Partial order

<u>Gigascope</u>

```
Select count(*)
From      A, B
Where     A.a-1 <= B.b and
          A.a+1 > B.b
A.a, B.b
monotonically increase
```



Absolute time model

- Information items have an associated timestamp
- Defining a single point in time w.r.t. a (logically) unique clock
  - Total order
- Timestamps are fully exposed to the language
- Information items can be timestamped at source or entering the engine

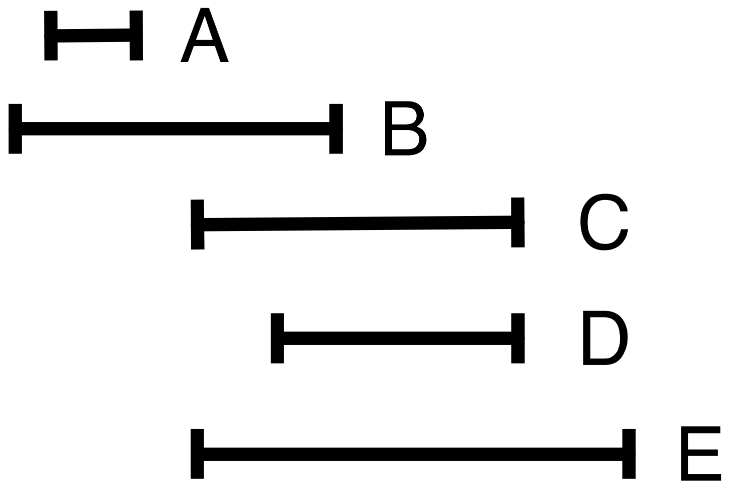
```
TESLA/T-Rex
Define Fire(area: string, measuredTemp: double)
From   Smoke(area=$a) and last
       Temp(area=$a and value>45) within 5 min. from Smoke
Where  area=Smoke.area and measuredTemp=Temp.value
```

Interval time model

- Used for events to include “duration”
- At a first sight, it is a simple extension of the absolute time model
  - Timestamps with two values:
  - Start time and end time
- However, it opens many issues
  - What is the successor of an event?
  - What is the timestamp associated to a composite event?

Interval time model

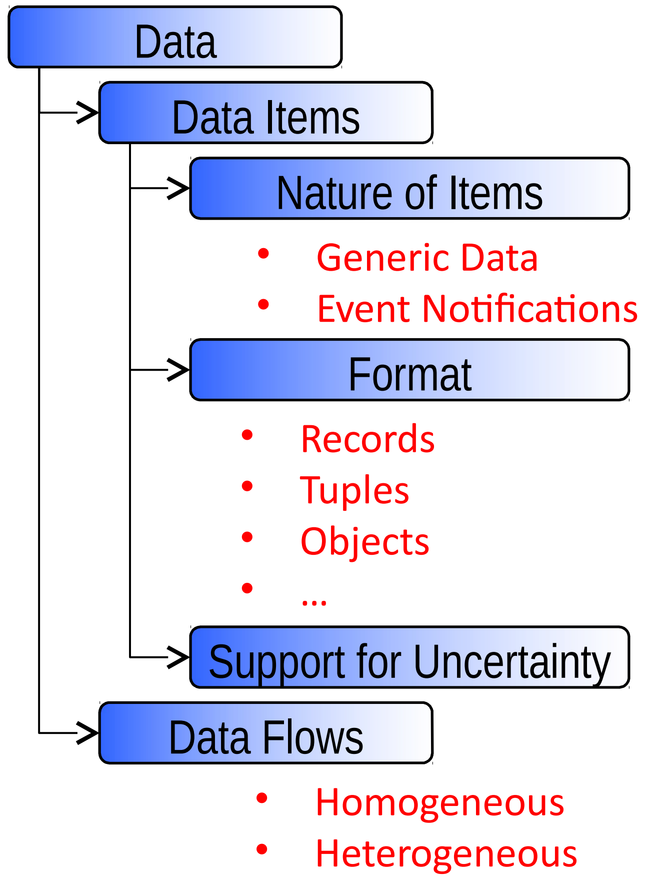
- Which is the immediate successor of A?
  - Choose according to end time only: B
    - But it started before A!
  - Exclude B: C, D
    - Both of them?
    - Which of them?
  - No other event strictly between A and its successor: C, D, E
    - Seems a natural definition
    - Unfortunately we loose associativity!
      - X✉(Y✉Z) ≠ (X ✉Y)✉Z
    - May prevent rule rewriting for processing optimizations



Interval time model

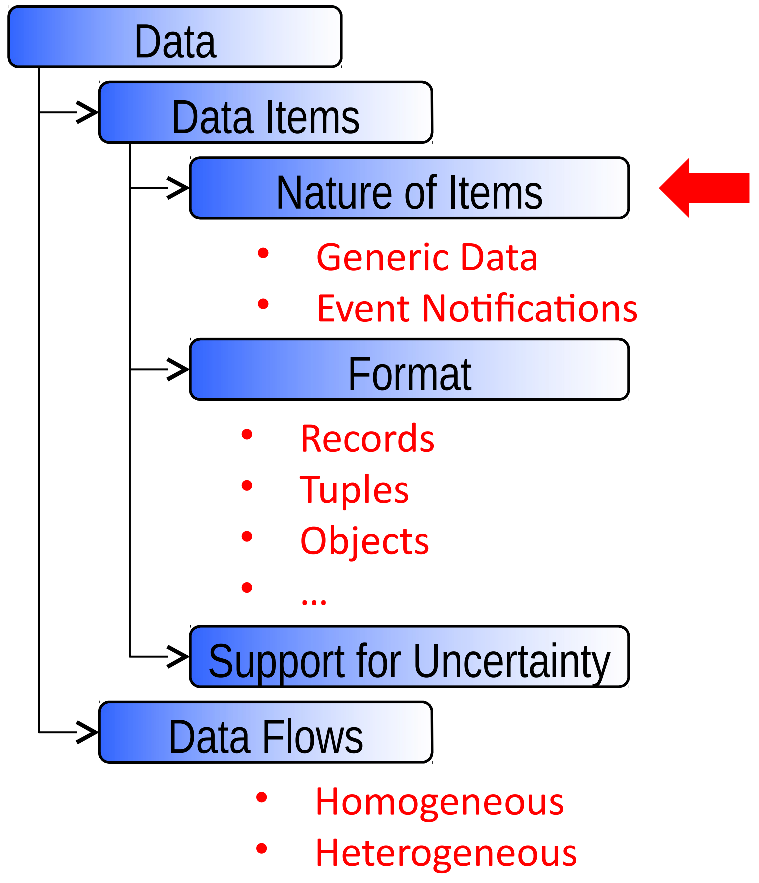
- *What is "next" in event processing?* by White et. Al
  - Proposes a number of desired properties to be satisfied by the "Next" function
- There is one model that satisfies them all
  - Complete History
- It is not sufficient to encode timestamps using a couple of values
  - Timestamps of composite events must embed the timestamps of all the events that led to their occurrence
  - Possibly, timestamps of unbounded size
    - In case of unbounded Seq

Data model



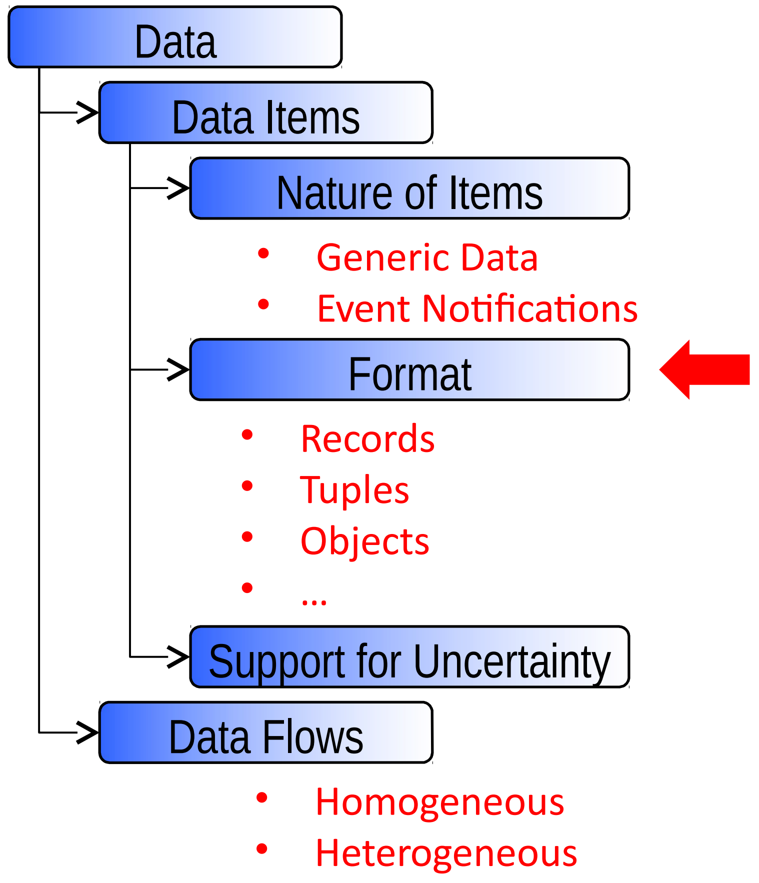
- Studies how the different systems
  - Represent single data items
  - Organize them into data flows

Nature of items



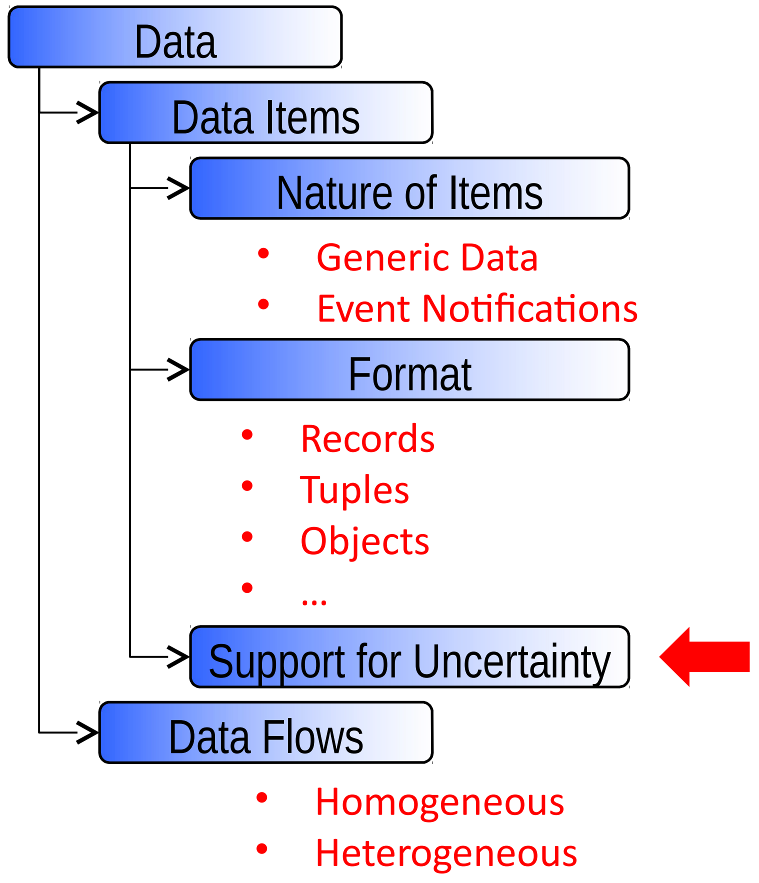
- The meaning we associate to information items
  - Generic data
  - Event notifications
- Deeply influences several other aspects of an IFP system
  - Time model !!!
  - Rule language
  - Semantics of processing

Format of items



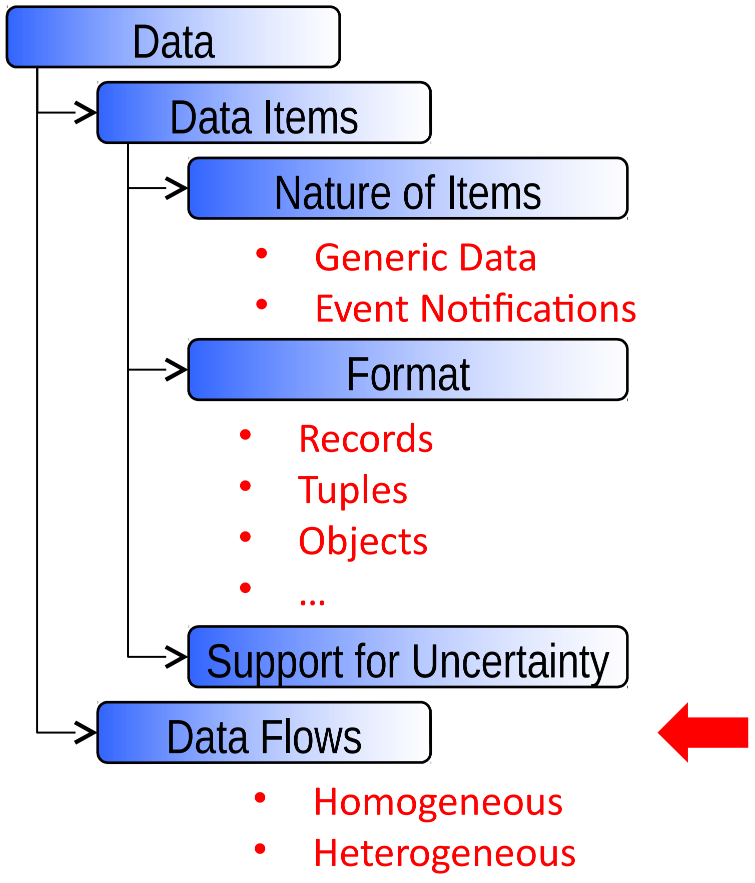
- How information is represented
- Influences the way items are processed
  - In DSMS, the relational model requires tuples
  - In RP, streams are often typed to enable integration with the programming language type system

Support for uncertainty



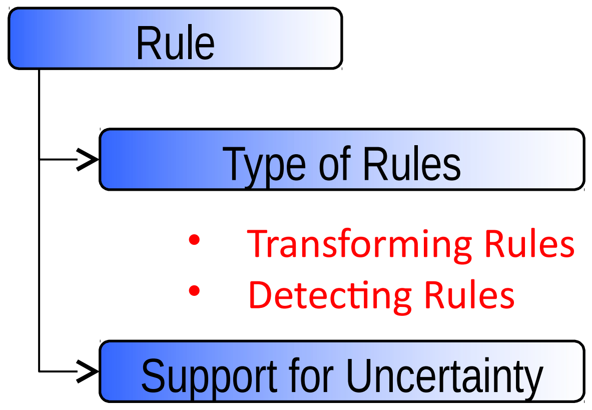
- Ability to associate a degree of uncertainty to information items
- To the content of items
  - Imprecise temperature reading
- To the presence of an item (occurrence of an event)
  - Spurious RFID reading
- When present, probabilistic information is usually exploited in rules during processing

Data flows



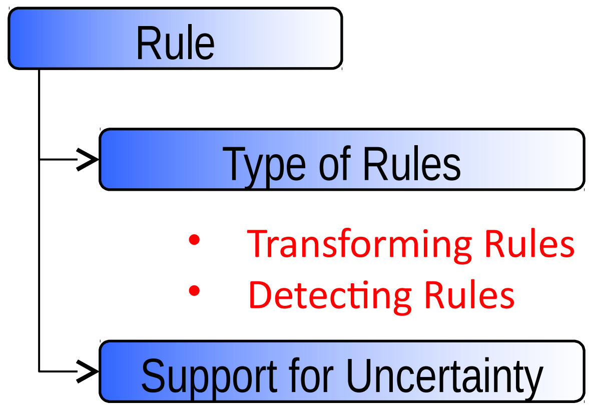
- Homogeneous
  - Each flow contains data with the same format and "type"
  - E.g. Tuples with identical structure
- Heterogeneous
  - Information flows are seen as channels connecting sources, processors, and sinks
  - Each channel may transport items with different kind and format

Rule model



- Rules are much more complex entities than data items
  - Large number of different approaches
  - Already observed in the previous slides
- We classify them into two macro classes
  - Transforming rules
  - Detecting rules

Support for uncertainty



- Two orthogonal aspects
- Support for uncertain input
  - Allows rules to deal with/reason about uncertain input data
- Support for uncertain output
  - Allows rules to associate a degree of uncertainty to the output produced

Language model

- Following the rule model, we define two classes of languages:
  - Transforming languages
    - Declarative languages
    - Dataflow languages
      - Functional and/or imperative operators
  - Detecting languages
    - Pattern-based

Declarative languages

- Specify operations to transform input flows to produce one or more output flows

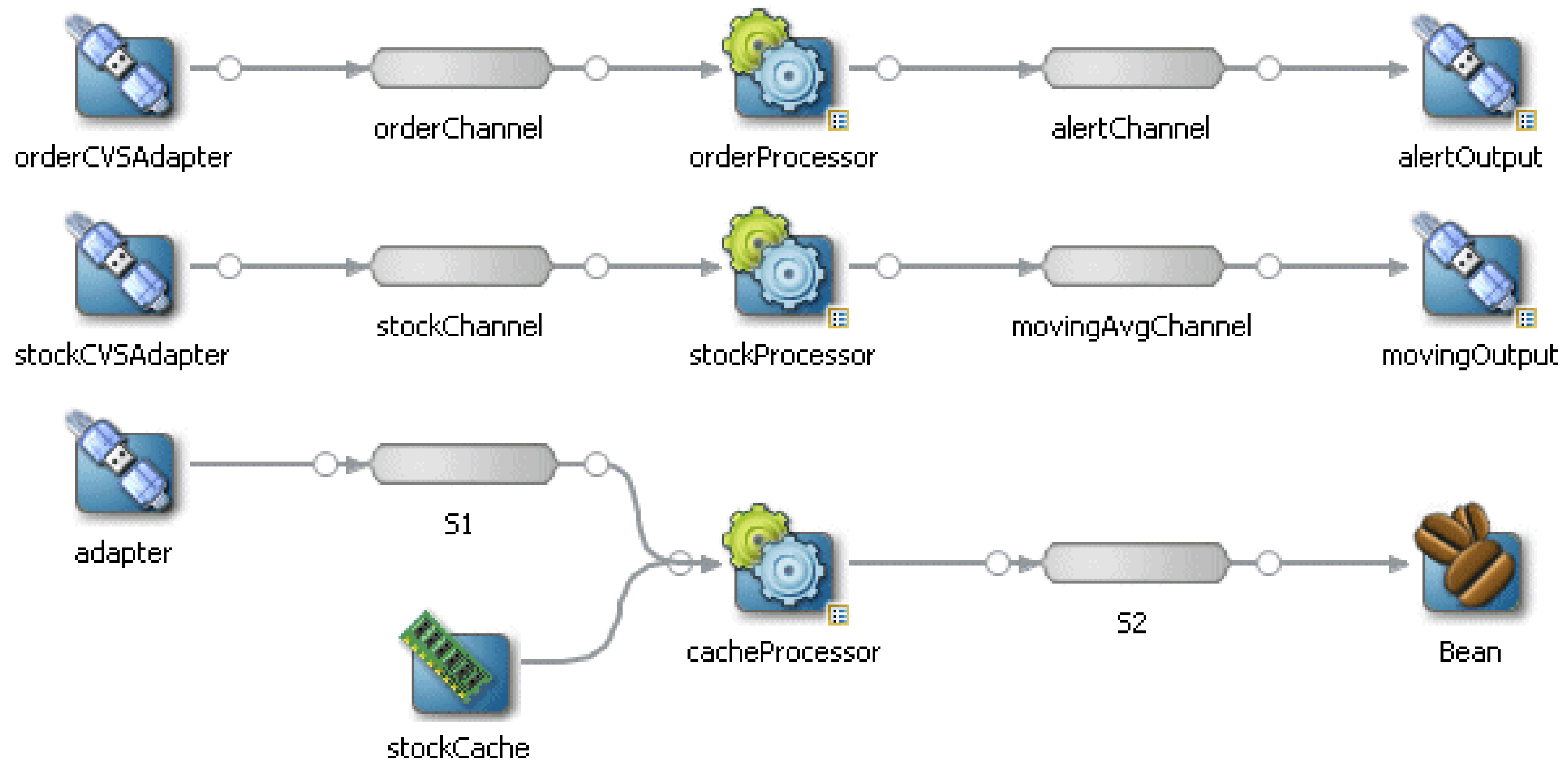
- Two main flavors
  - Relational (DSMS)
    - Select, join, aggregate operators
    - Windowing operators to select portions of the stream
  - Functional (RP)
    - Map, reduce, filter
    - Rare use of windowing operators

Dataflow languages

- Specify the desired execution flow
  - Starting from primitive operators
  - Example: Oracle CEP, Storm
- Can be user-defined
- Usually adopt a graphical notation

Imperative languages

Oracle CEP



Declarative languages

- Specify a firing condition as a pattern
- Select a portion of incoming flows through
  - Logic operators
  - Content / timing constraints
- The action uses selected items to produce new knowledge

Detecting Languages

<u>TESLA / T-Rex</u>

ACTION

```
Define Fire(area: string, measuredTemp: double)
From   Smoke(area=$a) and last
    Temp(area=$a and value>45)
    within 5 min. from Smoke
Where  area=Smoke.area and
    measuredTemp=Temp.value
```

CONDITION (PATTERN)