Low-Latency Service Data Aggregation using Policy Obligations

Stephan Reiff-Marganiec *, Marcel Tilly †, Helge Janicke ‡
* University of Leicester, Leicester, UK
Email: srm13@le.ac.uk
† European Microsoft Innovation Centre, Munich, Germany
Email: marcel.tilly@microsoft.com
‡ De Montfort University, Leicester, UK
Email: heljanic@dmu.ac.uk

***Abstract*—The Internet of Things, large scale sensor networks or even in social media, are now well established and their use is growing daily. Usage scenarios in these fields highlight the requirement to process, procure, and provide information with almost zero latency. This work is introducing new concepts for enabling fast communication by limiting information flow through filtering concepts combined with data processing techniques adopted from complex event processing. Specifically we introduce a novel mediation services architecture using filter policies to reduce latency. The filter policies define when and what data services need to provide to the mediator and thus save on bandwidth. The filter policies describe temporal conditions between two events removing the need to keep a complete history while still allowing temporal reasoning. Promising experimental results highlight the advantages to be gained from the approach.**

I. Introduction

The vision of the Web has changed over the past few years. This is triggered by several aspects, such as (1) the paradigm shift to services, (2) an increasing use of mobile devices and (3) new intelligent objects providing continuous access to data. Besides classical static web pages there are services and other data sources, such as sensors and devices. The term "Internet of Things" was coined, reflecting the trend that more and more data sources and services will be available in the future to provide a wider variety and broader set of new information, such as environmental information, geo location or social interactions. On the other hand there is an increasing demand from users to have access to information from all kinds of different devices (phones, PC, etc.). This information needs to be available in close to real-time in order to reply to requests, make decisions and stay competitive. This adds new requirements to middleware handling these devices and information, caused by a huge data volume produced by a large number of sources that need to be procured, processed and provided with almost zero latency. We should note that the huge data volume does not arise (at least in the scenarios we consider and address) from large single items of data that occur in scientific computing but rather from a very large amount of small items such as sensor readings.

We will now consider a fleet management system as a motivating example, however the approach is not limited to this scenario and can be applied in a wide variety of applications where services are selected from a large set of potential providers, such as sensor network, logistics, industry, military or consumer space. In fleet management, like taxi companies, with a large amount of taxis it is almost impossible to use the classical request-response approach to find the nearest taxi for a given user location. Therefore, the fleet management must be aware of the taxis location at any given time. The management system only requires the latest data to process a user request to locate the nearest taxi, thus there is no necessity to persist the data for later use. In the scenario (see Figure 1) there is a customer with a given context, his geo location, requesting a taxi. The fleet management system has to identify the most relevant taxi in terms of (1) availability and (2) proximity to the customer's location. There are two taxis, A and C, which are close to the customer's location, but they are not available. Taxi B is the closest which is available. Of course the fleet management could take traffic information into account, and then maybe taxi D becomes the best solution because it is reasonably close, available and might arrive earlier because of beneficial traffic conditions.

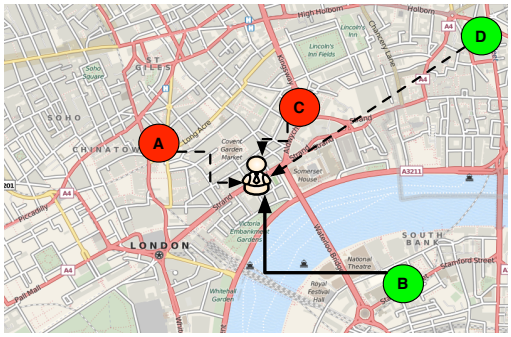

Fig. 1: Fleet Management Sample

This scenario shows (1) how different kind of properties of taxis (here: availability and geo location), (2) properties of different services (here: taxi and traffic) are used to select services, and (3) that taxis have to pro-actively inform the fleet manager about their location to enable fast and reliable responses to customer requests. Furthermore, the geo location and the traffic information are data which changes rapidly and it does not make sense to store all of this data because it is only short-lived and hence only the current values are relevant when a service has to be selected.

To achieve almost zero latency data processing, data must

be available at the place where the user needs it. So, instead of pulling data at request time from data sources or services, data should be pushed to a middle layer (here mediator). This is only the first step towards a faster processing of data in terms of providing results with low-latency. If the data sources are continuously pushing data to the mediator there is a vast amount of overhead by unnecessarily transferring data – a waste of bandwidth. The mediator informs the sources under which changing situation (when) the sources should inform the mediator about the change of their properties (what). What and when can be expressed with policy obligations which are injected into the data sources, so that we can really make use of their intelligence. Thus, each data source will be responsible to make the projection from its own fine-grained, raw data to some more high-level, complex data the mediator – and ultimately the user – is interested in. The obligations can be as smart as possible by using various sets of information, such as the prioritization of the data. Consider for example an alarm situation with cascading alarms. Such a system has to ensure that the most severe alarms are delivered and the bandwidth is not occupied with unimportant information. Thus, policy obligations executed on smart data sources – intelligent objects or services – should enable low capacity filtering by being context-aware.

In this paper we provide an approach to overcome the problem of procurement, processing and provision of information in real-time in combination with optimising the data traffic. We use concepts adopted from complex event processing to enable a real-time view of service properties to enable a fast and accurate view of their values with an application in real-time service selection. Initial ideas from [20] and [21] are refined in this paper, and we add a clear definition of the service properties as a projection of the policy obligation injected on data sources, such as services. The novel contributions of the paper are (1) a middleware architecture, data model and selection process to put the above into practice, and (2) a clear and well founded definition of policy obligations using context information expressed at an abstract XML level as well as formally through ITL (Interval Temporal Logic). Our approach can be seamlessly integrated with existing service selection approaches.

Section II provides some essential background work. Sections III and IV are the core of the paper and present the architecture and selection process respectively. Section V provides a formal description of the policy-based filtering approach based on temporal projection. Section VI provides an extended example to demonstrate the approach and section VII presents discusses experimental results. Section VIII points to some related work while section IX concludes the paper and considers further work.

II. Background

This section introduces the basic ideas which we combine to improve service selection and mediation approaches for consumer requests in real-time. As we formally model the temporal abstraction, we will also provide a short introduction to interval temporal logic.

A. Non-functional Properties

Non-functional and functional properties are used for service selection or context-based service discovery. Typically the properties are pulled from service repositories (that is from service metadata) or possibly from the services directly before an algorithm determines the most relevant service for a given context. Repositories are useful for static data and polling services directly works for small numbers of properties of a small number of services. The examples highlighted an emergent need to enable the continuous evaluation of functional and non-functional properties with a large number of services. We define static properties *ps* as constant over time (e.g. the location of a printer) and dynamic properties *pd* are changing over time (e.g. the length of a print queue). Non-functional properties NFP are defined as a tuple of static and dynamic properties: $NFP(t) = \langle ps, pd \rangle$

For the fleet management scenario the schema of the non-functional properties might look as follows:

```
<NFProperties>
  <Static>
        <TaxiId type="xs:string"/>
  </Static>
  <Dynamic>
        <GEOLocation id="x">
          <Longitude type="xs:int"/>
          <Latitude type="xs:int"/>
        </GEOLocation>
        <PassengerNumber type="xs:int"/>
  </Dynamic>
</NFProperties>
```

This presents a snapshot in time, with temporal aspects covered by events: we would see different data at different points in time.

Policies refer to obligations placed on a services to actively communicate *dynamic* information, with respect to a given data-schema, triggered by events and time. Informally this means that a policy defines the granularity over time at which data is pushed up the service chain to aggregating services and end-users. In this work the policies are modelled similar to the well-understood Event-Condition-Action paradigm [9], [23]. However, the novelty of the policies used in this work is that they use temporal conditions that describe the *distance* between two consecutive actions that push data to aggregating services, rather than defining condition on the system state. The advantage of this approach, compared to existing temporal conditions [13], [11], is that the condition bridges between two events, thus does not require the storage of large amounts of historical data. Informally a policy is a set of rules of the following structure:

```
<Policy> <!-- send to Service -->
  <Rule>
    <Target>...</Target>
    <Event>...</Event>
    <Condition>...</Condition>
    <Action>...</Action>
  </Rule>
  <Rule> ... </Rule>
</Policy>
```

The `<Target>` of a rule is a list of services on which the `<Action>` of the rule is invoked if the rule is triggered. The

<Event> of a rule is an event descriptor that determines when the <Condition> of the rule is evaluated. The descriptor is a predicate build from primitive domain dependent events (e.g. a GPS-Update) defined in the service description. Conceptually the event descriptor describes an abstraction of the event trace over which the <Condition> is evaluated. The <Condition> describes the distance between events that are communicated upstream to aggregating services as a temporal formula. The syntax that is used is an XML representation of Interval Temporal Logic formulae described next.

B. ITL

ITL [5] is based on the concept of an *interval* which is an (in)finite sequence of states $\sigma_0,\ \sigma_1 \ldots$. Each state σ_i maps from the set of variables Var to the set of values Val. The length $|\sigma|$ is one less than the number of states in the interval. The syntax of ITL is given in Figure 2 where μ is a constant value, a is a static variable (does not change within an interval), A is a state variable (can change within an interval), v a static or state variable, g is a function symbol and p is a predicate symbol. The syntax is based on [5], however uses the projection operator $f_1 \Delta f_2$ as primitive and derives the operator f^* as introduced in [16].

Expressions	
$e ::=$	$\mu \mid a \mid A \mid g(e_1, \ldots, e_n) \mid \bigcirc v \mid \mathsf{fin}\ v$
Formulae	
$f ::=$	$p(e_1, \ldots, e_n) \mid \neg f \mid f_1 \wedge f_2 \mid \forall v \bullet f \mid$ $\mathsf{skip} \mid f_1 ; f_2 \mid f_1 \Delta f_2$

Fig. 2: Syntax of ITL

The informal semantics of the most interesting constructs are as follows:

- skip: unit interval (length 1, i.e., an interval of two states).
- $f_1 ; f_2$: ("chop") holds if the interval can be decomposed ("chopped") into a prefix and suffix interval, such that f_1 holds over the prefix and f_2 over the suffix, or if the interval is infinite and f_1 holds for that interval. Note the last state of the interval over which f_1 holds is shared with the interval over which f_2 holds.
- $f_1 \Delta f_2$: ("projection") is defined to be true on an interval σ iff two conditions are met. First, the formula f_2 must be true on some interval σ' obtained by projecting some states from σ. Second, the formula f_1 must be true on each of the subintervals of σ bridging the gaps between the projected states.
  In the interval σ the value of K increases from 0 to 8 in steps of one. The interval σ satisfies $(\mathsf{len}(2))\Delta(K\ \mathsf{gets}\ K+2)$. $(\mathsf{len}(2))$ is true if the interval is of length two and $(K\ \mathsf{gets}\ K+2)$ is true if the K increases by 2 from state to state. The gaps between the projected states (highlighted in red) are bridged by the formula $\mathsf{len}(2)$. The formal definition of this operator is given in [16].
- $\bigcirc v$: value of v in the next state when evaluated on an interval of length at least one, otherwise an arbitrary value.
- fin v: value of v in the final state when evaluated on a finite interval, otherwise an arbitrary value.

We also introduce a number of derived constructs, but here we only show the subset directly used in this paper (more details are available in [5]). The binary operators $\vee$ (or) and $\supset$ (implication) are derived as usual.

$\bigcirc f \ \widehat{=}\ \mathsf{skip} ; f$ (read "next f"), means that f holds from the next state; $\mathsf{more} \ \widehat{=}\ \bigcirc\mathsf{true}$ means the non-empty interval; $\mathsf{empty} \ \widehat{=}\ \neg\,\mathsf{more}$ means the empty interval; $\mathsf{halt}\ f \ \widehat{=}\ \Box(\mathsf{empty} \equiv f)$ means terminate the interval when f holds; $v\ \mathsf{gets}\ e \ \widehat{=}\ \Box(\mathsf{more} \supset (\bigcirc v) = e)$ assigns v the value of e evaluated in the previous step (except the initial state); and $f^* \ \widehat{=}\ f\Delta\mathsf{true}]$ (read "f chopstar") holds if the interval is decomposable into a finite number of intervals such that for each of them f holds, or the interval is infinite and can be decomposed into an infinite number of finite intervals for which f holds. Also,

$$\mathsf{len}(e) \ \widehat{=}\ \begin{cases} \mathsf{false} & \text{if } e < 0 \\ \mathsf{empty} & \text{if } e = 0 \\ \mathsf{skip} ; \mathsf{len}(e-1) & \text{if } e > 0 \end{cases}$$

holds if the interval length is e;

III. Basic Concepts

It is quite challenging to obtain an accurate view of data coming from a huge number of different sources (here: services) with classic request-response approaches which are usually employed in SoC. Consider the number of printers within a company or all taxis of a company within a city. The number of possible services is high. In addition the dynamic properties such as the length of the print queue or the geo location of taxis change very frequently. Using a typical request-response approach every time a user asks for a taxi the system has to poll all the taxis' geo locations and other properties – this polling approach cannot scale and as a consequence a consumer will not get a reasonable response to his request. In such realistic settings it is becoming quite challenging to answer a simple question such as "find the nearest taxi to my location" quickly.

We can define this more crisply as a need for an concept delivering responses with low latency based on dynamic service properties at any time to consumer requests from huge lists of services. Basically, we propose to combine existing request-response approaches with publish-subscribe techniques. Services offer dynamic properties to which consumer can subscribe, such as the dynamic GeoLocation property of a taxis service and the number of current passengers from which the system can derive if the taxi is available or not.

We envision that our approach can be adopted easily as it only requires the addition of two interfaces: (1) The publisher endpoint is exposed on the service side to which the consumer can register or subscribe to events and (2) the subscriber endpoint is exposed by the *Mediator* to enable the services to fire events in a fire and forget fashion.

The publisher interface which enables the registry to subscribe to a set of dynamic properties provides two operations: $injectPolicy(policyObligation) : Id$ and $unsubscribe(Id)$. Here *policyObligation* describes the topic to be subscribed

to, the refresh interval, and the state changes which trigger event notification and *Id* is a unique registration id for the subscription. The subscriber interface offered by the *Mediator* provides only one operation, $notify(Event)$. An *Event* is a tuple of values $event = \langle se, ts, te, p \rangle$, containing the service endpoint address se, time information ts and te, and the payload p. The time information defines the valid start time ts and end time te of the event and the payload is defined by the type of the subscribed topic.

As described in [17] processing of streaming data is an important practical problem that arises in time-sensitive applications where the data must be analysed as soon as they arrive, or where the large volume of incoming data makes storing all data for future analysis impossible.

IV. Architecture

As a central instance we use a *Mediator* (see Figure 3). This *Mediator* encapsulates the processing of the incoming request from the consumer side and the incoming events from the service side and maps both. The *Mediator* is a service and exposed operations (methods) map internally to specific queries. Thus, during runtime the *Mediator* is receiving continuous streams of events from subscribed services. Incoming consumer requests are handled as a query on subscribed service properties. Instead of pulling data at request time the mediator knows at any time the status of all services. Therefore, this allows for service selection in real-time independent of the number of services.

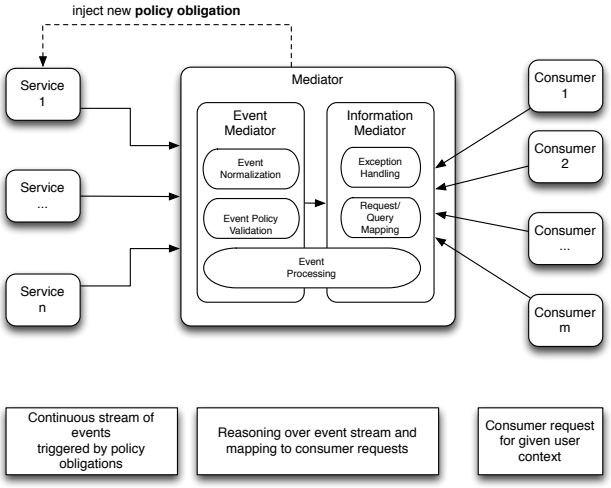

Fig. 3: Mediation service architecture

An event contains metadata (information about the time when the event was created on the publisher side) and a payload (the schema of the subscribed topic, such as temperature or vibration). New policy obligations are injected via the *Mediator* into the correct service (publisher).

The *Event Mediator* exposes an endpoint to collect all incoming events from registered services. Its responsibility is to normalize the incoming data streams. Usually, not all events provide the same data structure and therefore the Request Mediator maintains a mapping table to transform incoming events from endpoints into a normalised data stream. For example two services might provide temperature readings, but they use Celsius and Fahrenheit respectively. The *Event Mediator* normalizes event streams internally before forwarding data to the Information Mediator via the *Event Processing* component.

In addition the *Event Mediator* is able to detect missing events since the refresh time is set within the subscription process. It is possible to apply different retention policies to react to missing events, such as simply ignore missing events, use the latest event until a new event arrives, or raise an exception because the absence of an event is an exceptional case. How to handle missing events depends on the scenario and does not require a general solution.

The *Information Mediator* maps consumer request to queries on continuous event streams provided by the *Request Mediator*. On the consumer side the framework still offers a normal Web Service interface, which internally is transformed into a query executed over the event stream. The *Information Mediator* also ensures the quality of the events from event streams, such as duplicated events or out-of-order events. Here, our approach benefits from the existing work on complex event processing (CEP), such as [14] or [15]. Valid start and end times are generated by the service side but the *Information Mediator* added internal time information (System time) to the events. Clock increments internal to the *Information Mediator* move time forward decoupled from external sources, thus guaranteeing the order of events. (This idea is based on standard CEP technology, e.g. [3]).

To control the event flow from services to the mediator the services are accepting policy obligations as filtering rules. These obligations are defining which state changes within a service (on the source) trigger an event (such as "temperature $> 50.2C$") and the expected interval (refresh). The expected interval would then also be used within an event so that the start time is set when the event is issued on the service and the end time is defined by the refresh interval. Section V will deal with the specification of the required filters in detail. Being able to set the event interval rate and condition helps to fine-tune the system to obtain the appropriate balance between data accuracy, response time and data traffic.

V. Event Policies

We will introduce event policies by two representative examples. A simple example would reduce the number of events communicated upstream:

Example 1 (Simple Event Filter):

```
<Policy> <!-- send to Service -->
  <Rule>
    <Target>Aggregation Service</Target>
    <Event>GPSUpdate()</Event>
    <Condition>
      <LEN>2</LEN></Condition>
    <Action name="notify">
      <GPSLocation id="x"/></Action>
  </Rule></Policy>
```

The policy stipulates to send the *GEOLocation* on every *second GPSUpdate()* to the *Aggregation Service*. Another example would lead to an update being send to the aggregation service whenever the Euclidean distance between the last update and the current position exceeds 50m:

Example 2 (Update based on Dynamic Attributes):

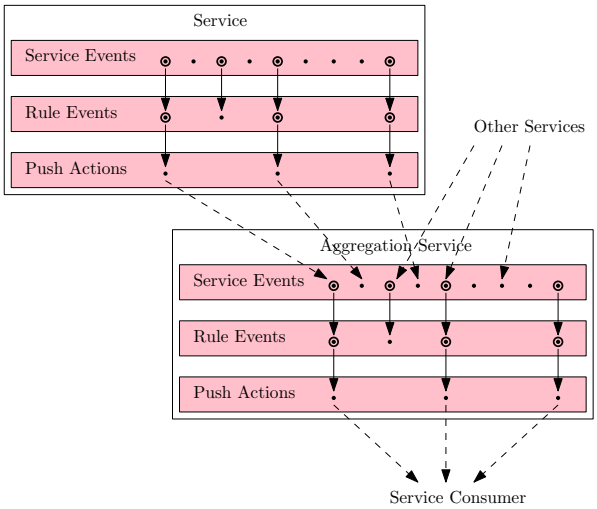

Fig. 4: Policy-Based Event Stream Filter

```
<Policy> <!-- send to Service -->
  <Rule>
    <Target>Aggregation Service</Target>
    <Event>GPSUpdate()</Event>
    <Condition><NEXT><HALT><GT>
          <FUNCTION name="EuclideanDistance">
            <GPSLocation id="x"/>
            <FIN> <GPSLocation id="x"/> </FIN>
          </FUNCTION>
          <CONST type="xs:int">50</CONST>
        </GT></HALT></NEXT></Condition>
    <Action name="notify">
      <GPSLocation id="x"/></Action>
  </Rule></Policy>
```

Building on earlier work [13], [11] in the context of access control, the condition of a policy is a temporal description of the filter that is applied to the selected event stream. Conceptually the service is filtering its event stream as depicted in Figure 4. Here the *Service* is processing its own stream of events (Service Events) that define its internal behaviour. The *Event* trigger in the definition of rules selects a sub-stream that contains only those states in the Service's behaviour at which the *Event* was raised. The condition in the rule is evaluated over this filtered event-stream and further restricts the behaviour based on the condition expressed in the rules. The condition is an interval temporal logic formula that defines the distance between any two selected states. In theses states an action is performed (e.g. "notify") that exposes information to other connected services, in this case the *Aggregation Service*. As a result, the policy determines the externally observable behaviour of the service. Connected services can influence this behaviour by updating the event policies via the `injectPolicy()` operation.

Services that support these policy filters can be combined into hierarchies, yielding service compositions that fuse and filter information defined in their NFP schemas based on policies. The information will be provided on a PUSH model with policies determining the frequency and conditions of updates, yielding a flexible, policy-based publish-subscribe infrastructure.

A. Formal Model

Let each service $s \in Services$ be defined over a continuous stream σ_s of events $e_i \in Events_s$, observed by the service s. This is modelled by representing σ_s as an ITL interval and $Event_s$ as a set of propositional state variables that indicate the occurrence of events (recall that state variables can change their value from state to state). This model allows for the concurrent occurrence of events, e.g. $e_i \wedge e_j$ $(i \neq j)$, and only captures the sequence of events, rather than their absolute timing. The creation time of the event is stored explicitly as part of the event tuple and can be referred to in the conditions of policy rules. As described in section III an event is described as a tuple $\langle se, ts, te, p \rangle$, denoting the service se creating the event, the time-stamp when the event was created ts, te (based on the clock of s) and an optional payload p. We use the notation $e.se, e.ts, e.te$ and $e.p$ when referring to a specific element of an event tuple e.

Evaluating the policy pol_s of the service s against this interval is a two stage process. In this way filtering of event streams based on simple events (evt_r) can be implemented very efficiently, and this reduction of events allows to significantly improve the evaluation of the conditions cnd_r, which is more complex and can in certain cases grow linearly with the number of states that are bridged.

1) Stage 1: For every rule $r \in pol_s$ an abstraction of the interval σ_s is generated based on the *Event* trigger evt_r of the rule r. Currently we only consider single event triggers, however the formal model is supporting combined events such as $e_i \wedge e_j$ or state formulae (i.e. ITL formulae that do not contain temporal operators). Conceptually this stage is generating an abstracted interval $\sigma_{s,r}$ of the interval σ_s that contains only those states in which evt_r is true. This is depicted in Figure 5.

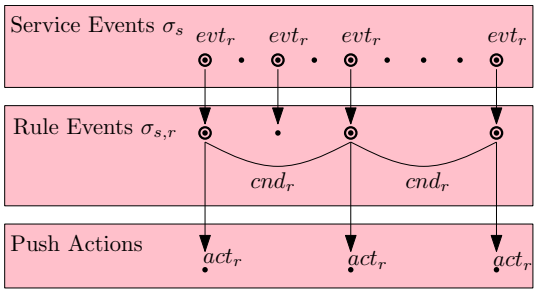

Fig. 5: Policy Rule Evaluation

2) Stage 2: For every rule r the condition of the rule cnd_r is evaluated against the corresponding abstracted interval $\sigma_{s,r}$. The condition defines the distance between two consecutive actions triggered by the same rule. This means that the temporal formula cnd_r must hold over the subintervals of $\sigma_{s,r}$ bridging the gaps between the projected states.

Formally this means that the policies relate the service's event trace, viz. the interval σ_s to actions that are performed by the service as follows: $\sigma_s \models \bigcirc\mathsf{halt}\,(evt_r)\Delta(cnd_r\Delta\Box act_r)$. Here $\bigcirc\mathsf{halt}\,(evt_r)\Delta f$ conceptually yields the abstracted interval $\sigma_{s,r}$ over which the policy rule is evaluated. The condition cnd_r of the rule then bridges between two consecutive actions that are performed as a consequence of the rule.

The overall service specification is then constructed from this (the specification of act_r is not detailed here): $\sigma_s \models \bigwedge_{r \in pol_s} \bigcirc\mathsf{halt}\,(evt_r)\Delta(cnd_r\Delta\Box act_r)$

The model can be implemented straightforwardly from its semantics using AnaTempura [10], [5], resulting in the following code for example 1:

```
/* run */ define example() = {
exists Evts :
{ /* create test event trace for the service */
  list(Evts,3) and stable(struct(Evts)) and evtmodel(Evts) and
  { /* example rule evaluation */
    (next halt(Evts[0]=1)) /* selecting events Evts[0] */
    proj{ /* show selected events, testing only */
      always format("Evts[0] = 1\n") and {
      len(2) /* select every second event only */
      proj{  /* show selected events, testing only */
        always format("Action on every 2nd Evts[0].\n")
}}}}}}.

set assign_ahead = false.
define evtmodel(Evts) = {
  Evts = [1,1,0] and skip ; Evts = [0,0,1] and skip ;
  Evts = [1,1,0] and skip ; Evts = [1,0,1] and skip ;
  Evts = [0,0,0] and skip ; Evts = [0,0,1] and skip ;
  Evts = [1,0,0] and skip ; Evts = [1,0,0] and empty
}.
```

Here three events are modelled for the service, and an example trace is generated by the function `evtmodel(Evts)`. AnaTempura can be run in a run-time verification mode and could receive these events from an external program. The event trigger for the encoded rule is `Evts[0]`, where a value of `1` indicates that the event occurred. This is encoded in the first projection condition `(next halt(Evts[0]=1))`, which in effect generates the more abstract interval $\sigma_{s,r}$ over which the second projection is taking place. In this example the temporal condition is selecting every second of the events (`len(2)`) on which the action of the rule is triggered. In this proof of concept only a statement is printed out to the screen, but instead a message could be easily send to another service. The above code can be readily executed in AnaTempura (available at http://www.cse.dmu.ac.uk/STRL/ITL/).

The advantage of the formal model is that one can reason about the hierarchy of event filters throughout the service infrastructure. Without loss of generality one can reason about a general stream of events E_{sys} that contains all events that are observable in the system. Whilst the event streams E_s are generated by the individual services s, conceptually they can be seen as a filtered event stream that selects from E_{sys} only those events that originate from s. This approach makes reasoning about the interaction of the various event streams possible and does not complicate the analysis as it uses the same policy-defined event filters that are advocated in this work.

VI. Example (cont'd)

Let us go back to the taxi management scenario to illustrate the presented theory with a simple example. In this example users are interested in finding a taxi. They formulate their needs in form of a request providing their location. The *Mediator* is getting this request and is responsible for mapping the user request to the available event streams coming from all taxis. Here it is a query returning the closest available taxi.

The user requests are joined with the event stream coming from the taxis using temporal join-statements expressed through SQL-like expressions. For example Microsoft StreamInsight [1] code would be as follows :

[1] http://msdn.microsoft.com/en-us/library/cc160860.aspx

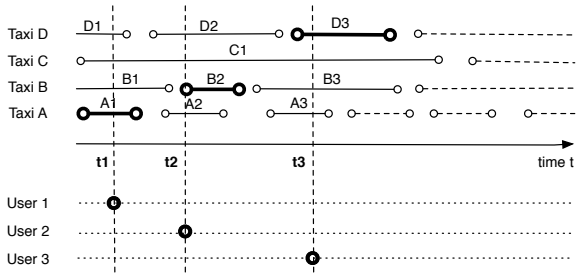

Fig. 6: Event Stream Reasoning

```
/* define incoming streams */
var taxiStream = CEPStream<Taxi>(...);
var userStream = CEPStream<User>(...);
/* run standing query */
var result = FROM taxi IN taxiStream
             JOIN user IN userStream
             WHERE (taxi.Available
                AND Min(Distance(User.GEOPosition, taxi.GEOPosition)))
             SELECT taxi; /* create a result stream of relevant nodes
```

Each taxi is sending an event to the *Mediator* when it moves more than 50 meters or the number of passengers is changing (Example 1). A so-called standing query consumes the incoming event streams from all taxis and maps the user request to the stream. The *Mediator* is aware of all state changes and hence aware of the state of the entire system and thus can reply to the request with almost zero latency.

Let's assume we have four taxis (A,B,C, and D) (see Figure 6). At time t1 a user is requesting a taxi. At that point in time only taxi A is available so that the query will return taxi A as result. Thus, taxi A is sending a new event (A2) since it picks up the user. A new user requesting a taxi at time t2 encounters a new situation with taxi B and taxi D being available. The query will check if B or D is closest to user2.

Already this simple scenario highlights the low-latency data processing capabilities of this approach.

VII. Experimental Results

The mediator approach with filtering of events at the source was developed to address two key problems, namely (1) a need to provide replies with near zero latency and (2) a requirement to reduce the amount of data transfer (recall that this was large because of the amount of small messages). Validation was geared towards proving these two aspects, so we conducted two evaluations: (1) we measured the latency of finding a result using the pull model compared with a push model and (2) counted the number of messages occuring in a one second time interval in the push and combined pull-push model. We considered settings with up to 60000 services.

Testing pull and push approaches is quite complex since there is a big number of data sources required to get some significant results. There is also a need to distinguish between pure latency as a result of the different concepts and system or network latency and latency used by semantic processing. Thus, for testing we decided to run everything on one machine and to simulate each service as a small object. Due to this, latency caused by the physical setup can be neglected. The simulated services were holding a random value and the query

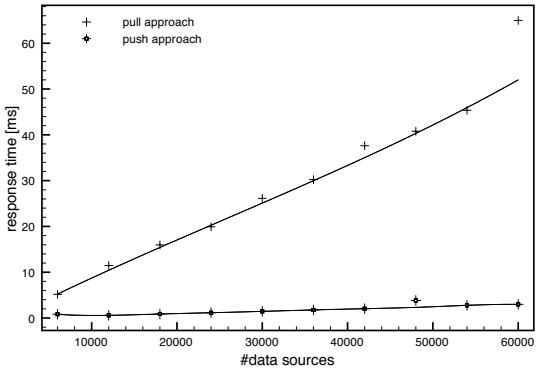

Fig. 7: Latency of pull approach vs. push approach

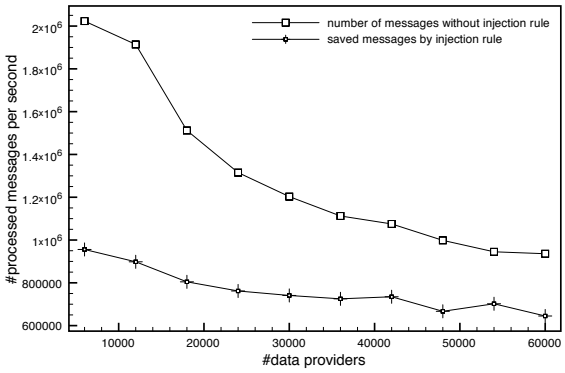

Fig. 8: Push approach with injected policy and without

was to find the object whose value is the closest to a given number (in the taxi example the value would represent the taxi position and the given number the customer position).

In the pull approach we are iterating over all objects (*taxis*) and are trying to find $min(value - number)$. In the push approach the objects are pushing their number to the mediator and we run a query over this data.

The results are presented in Figure 7. While the latency is increasing linearly for the pull approach, it remains almost constant for the push approach. Also, all values for the push approach are far lower than those for the pull approach, with for example approx 4ms vs 65ms for 60000 services. This clearly indicates that the pull approach is superior to the push approach in terms of latency. The trends show that for a growing number of services we can deliver performance close to no latency. We also want to point out that this is the pure meassured latency ignoring network and processing delays – these will make latency become rapidly worse for the pull approach as much more communication and more processing is needed compared to push approach which remains constant for the full spectrum (albeit a little slower in real terms than meassured in the isolated setting).

The drawback of push approaches is the number of data items sent; we have introduced policies in our approach to avoid that messages are polluting the network unnecessarily. The second validation is comparing the number of messages send in a one second interval in a pure push approach compared with the number of messages which are sent in a push approach with filtering. The latter is expected to send fewer messages because of the injected policy. For testing we extended the objects with another random value representing the availability, either 0 or 1. The policy is saying that the object should push only messages when the availability is 1 (means the taxi is available). Figure 8 shows the results as we expected. With a trivial policy we can save around 50% of exchanged messages (based on the randomly changing values). More specific policies (*such as taxi should have moved more then 50 meters before sending an update*) and real data (a specific taxi being available 50% of the time does not make for a profitable business model!) the number of messages can be further reduced.

Overall, both simple tests highlight how (1) using the push approach with mediator and (2) policy injection on the data source can be combined to form a promising architecture supporting low-latency for large systems.

VIII. Related Work

While there is work on non-functional properties, service selection, complex even processing and temporal logic there is no work as far as we are aware which combines these approaches to solve selection problems for large scale systems. [4] describes a very interesting approach for using active rules for pushing reactive services. But it does not take into account temporal aspects or states. [18] presents a framework for satisfaction of complex data needs involving volatile data. The focus is on pull-based environments; we believe that our approach is more promising for large scale systems. Push based system research focuses mainly on aspects of efficient data processing, where load shedding techniques [22] can be applied in order to control what portion of the data to process. Such systems include publish-subscribe (pub/sub) ([7]), stream processing ([1]), and complex event processing, however there is no consideration of bandwidth consumption.

As mentioned there is much work on service selection based on NFPs, a survey can be found in [24]. Most work concentrates on defining QoS (Quality of Service) ontology languages and vocabularies and identification of various QoS metrics and their measurements with respect to semantic services. All these approaches are lacking temporal aspect.

The use of Event-Condition-Action (ECA) rules is well established in Data-Stream Processing applications e.g. [6] and ECA-based policy languages e.g. [23] are used to control and manage distributed systems [19]. In our work, we are however mainly concerned about the selection and propagation of events. Our formalisation of event policies differs from traditional ECA rules in that the condition does not only describe a Boolean combination of events, but can address the history of a selected event stream allowing to specify the *distance* between propagated events using ITL [5]. Data Stream processors such as SNOOP [6] and successors already use event histories for detecting the order of events, making this a natural model for expressing policies that also allows for the efficient enforcement of such policies [12].

The use of policies together with a mediator has been suggested in a different context in [8] with a focus on mining transaction data for fraud detection. Their use of policies is

targeted to this particular application domain and is focused on the detection of events.

The work that is possibly closest to ours is [2], which introduces a framework for providing information based on sensors to mobile users. The work is in some way complimentary to ours, in that it proposes an approach that registers triggers based on user demands in the SLIM Service. Data sources report their data to the SLIM Service and the service reports data back to users when the trigger conditions are met. Our approach focuses on the reduction of traffic on obtaining the values from the data sources based on the registered filters, and then allows users to query the data in the mediator thus not requiring users to pre-register for concepts of interests.

IX. Conclusion and Future Work

We presented a new approach investigating service selection problems with a huge number of potential services and highly dynamic service properties. By combining NFP-based selection with ITL we have grounded dynamic properties on a valid formal model. We presented a way to use ITL to express policy obligations as ECA rules which should be executed close to the sources to enable an accurate view of a large scale system at any point in time and to reply to consumer requests with almost zero latency. The sources (in our example taxis) are notifying the mediation service about any state change defined by policies thus the mediation service can (1) reason about the incoming streams and reply immediately to consumer requests and (2) the mediation service can make assumptions in terms of missing data and forecast likely future behaviour.

So far, we have looked into the foundation aspects on how to tackle scenarios with a vast amount of data sources and data consumers. The next steps of our work will investigate more complex policies to be executed at the sources, such as the direction and speed of a taxi in addition to its position. We will also look into the prediction capabilities of our approach. Since the mediation service is aware of the lifetime of the information of each node it can look into the future to predict certain results. So far, the assumption is that the states are discrete. But we could also use functions for continuous states so that we have to address this problem during the selection process. In future work we will also further validate and compare our approach with others as well as consider alternatives approaches to ITL to explore whether computational gains can be made.

References

[1] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. Zdonik. Aurora: a data stream management system. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, pages 666–666, New York, NY, USA, 2003. ACM.

[2] Bhuvan Bamba, Kun-Lung Wu, Bugra Gedik, and Ling Liu. Slim: A scalable location-sensitive information monitoring service. In *ICWS 2013: IEEE 20th International Conference on Web Services*, pages 50–57, 2013.

[3] R.S. Barga, Jonathan Goldstein, Mohamed Ali, and Mingsheng Hong. Consistent streaming through time: A vision for event stream processing. *Arxiv preprint cs/0612115*, 2006.

[4] Angela Bonifati, Stefano Ceri, and Stefano Paraboschi. Pushing reactive services to XML repositories using active rules. *Computer Networks*, 39:645–660, 2002.

[5] A. Cau, B. Moszkowski, and H. Zedan. The ITL homepage: http://www.cse.dmu.ac.uk/STRL/ITL. Technical report, Software Technology Research Laboratory, De Montfort University, 2011.

[6] S. Chakravarthy and D. Mishra. Snoop: an expressive event specification language for active databases. *Data Knowl. Eng.*, 14:1–26, November 1994.

[7] A. Demers, J Gehrke, M Hong, M Riedewald, and W. White. Towards expressive publish/subscribe systems. *Advances in Database Technology-EDBT 2006*, pages 627–644, 2006.

[8] Michael Edge, Pedro Sampaio, Oliver Philpott, and Mohammed Choudhary. A policy distribution service for proactive fraud management over financial data streams. *Services Computing, IEEE International Conference on*, 2:31–38, 2008.

[9] Charles L. Forgy. Expert systems. In Peter G. Raeth, editor, *Expert systems*, chapter Rete: a fast algorithm for the many pattern/many object pattern match problem, pages 324–341. IEEE Computer Society Press, Los Alamitos, CA, USA, 1990.

[10] Roger William Stephen Hale. *Programming in Temporal Logic*. PhD thesis, Trinity College, University of Cambridge, October 1988.

[11] H. Janicke, A. Cau, F. Siewe, and H. Zedan. Dynamic access control policies: Specification and verification. *The Computer Journal*, 2012.

[12] Helge Janicke, Antonio Cau, François Siewe, and Hussein Zedan. Deriving Enforcement Mechanisms from Policies. In *Proceedings of the 8th IEEE international Workshop on Policies for Distributed Systems (POLICY2007)*, pages 161–170, June 2007.

[13] Helge Janicke, Antonio Cau, François Siewe, Hussein Zedan, and Kevin Jones. A Compositional Event & Time-based Policy Model. In *Proceedings of POLICY2006, London, Ontario, Canada*, pages 173–182, London, Ontario Canada, June 2006. IEEE Computer Society.

[14] Ling Liu, Calton Pu, and Wei Tang. Continual queries for internet scale event-driven information delivery. *Knowledge and Data Engineering, IEEE Transactions on*, 11(4):610–628, 2002.

[15] David Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman, Amsterdam, 2002.

[16] Ben Moszkowski. Compositional Reasoning about Projected and Infinite Time. In *Proceedings of the 1st IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'95)*, pages 238–245, Fort Lauderdale, Florida, Novenmber 1995. IEEE Computer Society Press.

[17] A Riabov and Z Liu. Scalable planning for distributed stream processing systems. In *Proceedings of ICAPS*, volume 06, 2006.

[18] Haggai Roitman, Avigdor Gal, and Louiqa Raschid. Web Monitoring 2.0: Crossing Streams to Satisfy Complex Data Needs. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 1215–1218. IEEE Computer Society, 2009.

[19] M. Sloman. Policy driven management for distributed systems. *Journal of Network and Systems Management*, 2:333–360, 1994.

[20] Marcel Tilly and Stephan Reiff-Marganiec. Matching customer requests to service offerings in real-time. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 456–461. ACM, 2011.

[21] Marcel Tilly, Stephan Reiff-Marganiec, and Helge Janicke. Efficient data processing for large scale cloud services. In *Proceedings of FM-S&C in Services 2012 the 2011 ACM Symposium on Applied Computing*. IEEE, 2012.

[22] Y.C. Tu, S. Liu, S. Prabhakar, and B. Yao. Load shedding in stream databases: a control-based approach. In *Proceedings of the 32nd international conference on Very large data bases*, pages 787–798. VLDB Endowment, 2006.

[23] K. Twidle, E. Lupu, N. Dulay, and M. Sloman. Ponder2 - a policy environment for autonomous pervasive systems. In *Policies for Distributed Systems and Networks, 2008. POLICY 2008. IEEE Workshop on*, pages 245 –246, june 2008.

[24] H.Q. Yu and Stephan Reiff-Marganiec. Non-functional property based service selection: A survey and classification of approaches. In *Proc. of 2nd Non Functional Properties and Service Level Agreements in SOC Workshop (NFPSLASOC'08)*. Citeseer, 2008.