Fast Data Processing for Large-Scale SOA and Event-Based Systems

Marcel Tilly
European Microsoft Innovation Center, Germany

Stephan Reiff-Marganiec
University of Leicester, UK

ABSTRACT

The deluge of intelligent objects that are providing continuous access to data and services on one hand and the demand of developers and consumers to handle these data on the other hand require us to think about new communication paradigms and middleware. In hyper-scale systems, such as in the Internet of Things, large scale sensor networks or even mobile networks, one emerging requirement is to process, procure, and provide information with almost zero latency. This work is introducing new concepts for a middleware to enable fast communication by limiting information flow with filtering concepts using policy obligations and combining data processing techniques adopted from complex event processing.

Keywords: Data Processing, SOA, Event-Based, Big Data, Fast Data, Internet of Things

INTRODUCTION

Today, there are various mega trends; people are talking about big data, cloud computing, service-oriented architecture (SOA), or the Internet of Things (IoT); just to name a few. All these trends have at least one common aspect: Data! There is a huge amount of data produced by a vast amount of heterogeneous sources, e.g. sensors, phones, cars, etc.. This data needs to be filtered, processed, and procured. Besides simply collecting all this data, there is rapidly growing demand to create timely insights into data. These insights can provide competitive avantages to business. Extracting relevant information from data or correlating data with other data sets as fast as possible is becoming a key factor for success. Latency, the time data needs to get processed, is getting more and more critical.

Some questions, which need to be answered, are:

- How can this data get processed as fast as possible?
- How can relevant data be separated from irrelevant data?
- How can data get filtered efficiently and scalable?
- How can data from distributed, heterogeneous data sources and services be integrated into a system?
- How to combine different technologies, different interaction patterns to make data flow efficient?

This paper is providing answers to these questions. To achieve almost zero latency data processing, data must be available at the place where the user needs it, such as a data provider. So, instead of pulling data at request time from data sources, data should be pushed to such a data provider. 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 a data provider (e.g. the selector) there is a vast amount of overhead by unnecessarily transferring data - a waste of bandwidth.

For mobile devices the cost of bandwidth needs to be taken into account. Let's assume the data provider, the one who interacts with the user, knows when the user needs updated data and the intelligent data sources know about their situation. Thus, the data provider informs the sources under which changing situation (when) the sources should inform the data provider about the change of their properties (what). What and when can be expressed with event policies, 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 data provider - and at the end 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, event policies executed on smart data sources - intelligent objects - should enable low capacity filtering by being context-aware.

There are already approaches available, which cover parts of the problem. Research has provided approaches to handle and process data with low latency, such as complex event processing (CEP). There are also approaches to distribute processing, such as the actor model. Most of the approaches are tackling only one specific aspect of big data, cloud computing or SOA. No approach is really trying to find a holistic answer to solve new mega trends, such as the Internet of Things.

The approach described in this paper is trying to combine promising approaches to enable fast processing of data in hyper-scale and distributed setups. Hyper-scale means that there are millions of data sources as we can find in IoT setups. Data sources here can be considered as services offering data. This data can change over time, such as the temperature offered by a temperature sensor. There are other services offering weather information or traffic information for example.

Our solution of combining the classical request-response paradigm with event-based approaches and technologies to process data and enabling insights with low-latency is described in this paper. Some parts of this work have been previously published ((Tilly & Reiff-Marganiec, 2011),(Marganiec, Tilly, & Janicke, 2014)), however in this paper our ideas, our different contributions and a new detailed view on the architecture are being brought together for the first time.

In combining existing paradigms, such as pub-sub approaches for processing service offerings and mediations with classical request-response SOA approaches for consumer requests facilitated by in memory data processing technologies, such as CEP or, more general, stream event processing, can help to overcome the afore mentioned challenges.

In addition it is worth to rethink the big data strategy to process all data in the cloud: Sometimes it makes more sense to process or aggregate data at the place where it is born.

By extending CEP queries and rules to be processed already on the service or device side, close to the sources, it is possible to reduce the amount of data, which needs to be send around. Queries can be used to correlate and aggregate data (events) at its origin as event policies. This approach can be used to overcome the scaling problem.

Mediation between consumer requests and service offerings is the most challenging one because it requires pattern mining and detection. Pattern mining and detection can be achieved by learning from interaction between users and services. As soon the pattern is learned it can be expressed in terms of a query on the stream of incoming data or can be pushed to the service side.

Although there is some relevant work around convergence of SOA and event-based systems, we think that our work goes behind this. Providing a concept for convergence of SOA and event-based is almost a side-effect. Our approach is going behind it since it targets the processing of data in such setups.

Our approach is also not about purely handling big data. It provides an approach to tackle setups which we typically find in IoT scenarios: huge amount of sources providing data which needs to be processed and procured. The fact that we have to deal with a huge amount of data is not the only one. We have to tackle aspects of where and how to process the data to achieve an optimized flow of data and create timely insights.

The presented arguments can be summarized into the following challenges:

- **Speed**: Speed means fast processing of data to provide timely insights. There is a demand on getting results as fast as possible. New paradigms are needed to improve the speed on processing service requests or in processing data provided by data sources, such as sensors. Basically, it makes sense to rethink classical request-response SOA approaches and to ensure that service offerings process data in almost real-time. This is a key challenge for moving forward towards to the next generation of the SOA.
- **Mediation**: Here mediation is used to describe the combination of request-response interaction pattern and event-based interaction. These different pattern needs to be combined and there must be concepts enabling their seamless integration.
- **Scale**: By combining interaction patterns the amount of transferred data will be increasing. Even more so if we consider sensor, cars or other data sources as service, which need to be integrated and their data be processed. This data mainly coming from event-based sources has to be optimized. Ideally a master service can control when data is forwarded and which data is forwarded through terms of aggregation or batching. A higher level of control of transferred data is required to optimize data traffic. This enables scaling up to millions of services sending data around.

The paper will provide a motivating example in section 2. Section 3 will give some insights into used technologies as background for section 4 which will introduce the core ideas of this work. The architecture and building blocks are described in section 5. Section 6 will show some experimental results. We conclude the paper with a look at related work and next steps.

MOTIVATING EXAMPLE

In real life there is plethora of example for hyper-scale setups, such as the connected car green wave, friend-finder or smart cities. We will illustrate some of these, before showing a working example that will be used throughout the paper.

In the connected car green wave scenario, car data and traffic signal data needs to be collected. The system will recommend to the driver the correct speed so that she can, depending on current traffic, catch the next signal light at green and does not need to stop. The current light signals state, the position and speed of the driver and the speed of the drivers nearby are relevant to find the correct speed. Clearly, some processing can happen in the cloud and maybe some processing in each drivers car. The calculation of the speed is the timely insight and has to happen ad-hoc and with low-latency otherwise the driver would not need it.

The friend-finder scenario is based on the user's social graph, their network of friends. Then the geo-positions in relation to geo-positions of friends can be considered. If one friend is near-by the system can inform the user. The processing can happen in the cloud, but also locally in the phone of the user to preserve privacy. In the privacy case the system would get only a region where the user is and would correlate this with the friends' regions. Considering this would run for all Facebook users we can easily talk about a hyper-scale system.

SmartCities is a mega trend and a meta scenario. There it is possible to find scenarios about traffic management or crime prediction. Each scenario for it requires collecting data and processing them. Some scenarios can be optimized in processing when data is filtered already at the data source. Some scenarios can help preserving privacy by not giving raw data about the user but by forwarding aggregated data and invite to the processing in a way that the hard raw data processing happens only on the device or service the user owns.

As a motivating example for the paper, we will now consider a fleet management system, 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 – namely the kind of situations discussed above. 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.

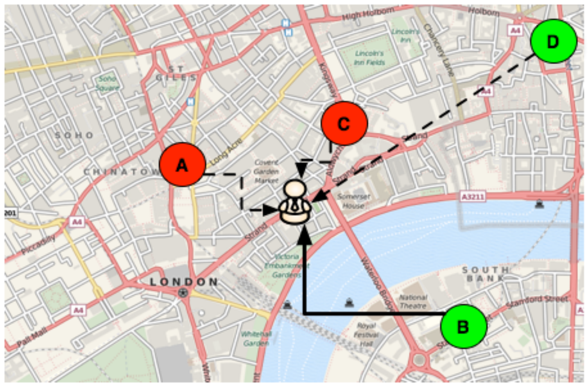


Figure 1: Fleet Management Sample

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 event policies, 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.

BACKGROUND

This section introduces the basic ideas, which we combine different approaches to enable data processing in a distributed setup. To clarify, this work is going beyond existing big data approaches since not the amount of data is the purpose of this work only but also the processing speed and an improved data traffic approach. There is a tendency to call this *fast data* instead of *big data* to address the challenges better, which occur in IoT setups for example.

Data processing in our scenarios considers data as time-stamped events as described in the motivating example. As we formally model the temporal abstraction, we will also provide a short introduction to interval temporal logic. The distribution and scaling aspect will be handled by making use of an actor model approach. Therefore we will provide some depth into the actor model as well. The same is true for complex event processing (CEP). We will provide some more depth insights into CEP as well.

However, to capture the state of related work we will start with on overview on most relevant work with regards to our approach. We will look into work around (1) rules and reactive systems, (2) data processing in regards of complex event processing, (3) services, service properties and service selection, and (4) ECA rules.

RELATED WORK

The use of Event-Condition-Action (ECA) rules is well established in Data-Stream Processing (Chakravarthy & Mishra, 1994) and ECA-based policy languages are used to govern the behavior of systems on the basis of these rules to control and manage distributed systems . In our work, we are however mainly concerned about the selection and propagation of events in a P2P infrastructure. The formalization of event policies in this work 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 that allow to specify the distance between propagated events using ITL (Cau, Moszkowski, & Zedan, 2011).

Data Stream processors such as SNOOP 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 (Helge Janicke, Cau, Siewe, & Zedan, 2007). The semantics of event-policies is based on temporal projection as this is a natural abstraction technique for complex system specifications. Other work by Duan et.al. (Duan & Koutny, 2004; Tian & Duan, 2009) on propositional projection temporal logic would provide alternative formalizations of projection, but lead to a more complex formalization of the event policies without apparent gain in this application context. The use of policies together with a mediator has also been suggested in a different context by Edge et.al. (Edge, Sampaio, Philpott, & Choudhary, 2008) where they focus on the mining institutional transaction data for fraudulent activities. However, their use of policies is targeted to this particular application domain and is focused on the detection of events, whereas we address the problem of event altering and propagation as part of an infrastructure for event driven P2P systems.

As already mentioned there is a lot of work about service selection based on non-functional properties. (H. Q. Q. Yu & Reiff-Marganiec, 2008) provides a survey and classification of service selection based on non-functional properties. Most of the related work on using non-functional properties for service selection 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. In (T. Yu, Zhang, & Lin, 2007) QoS ontology constraints for efficient service selection are described, while (Reiff-Marganiec, Yu, & Tilly, 2009) separates different non-functional criteria into different service categories. This is more sensible than ranking all kinds of services by using the same predefined criteria and hence not considering the different attributes that occur with specific services. All these approaches are lacking temporal aspect or NFPs.

Bonifati et al. (Bonifati, Ceri, & Paraboschi, 2002) describes a very interesting approach for using active rules for pushing reactive services. But it does not take into account temporal aspects or states. Roitman et al. (Roitman, Gal, & Raschid, 2009) presents a framework for satisfaction of complex data needs involving volatile data. But the focus is on pull-based environments. We believe that our approach is more promising for large-scale systems. With push-based systems, data is pushed to the system and the research focus is mainly on aspects of efficient data processing, where load shedding techniques (Tu, Liu, Prabhakar, & Yao, 2006) can be applied in order to control what portions of the pushed data to process and to increase latency. Such systems include publish-subscribe (pub/sub) (Demers, Gehrke, Hong, Riedewald, & White, 2006) stream processing (Abadi et al., 2003), and complex event processing, however there is no consideration of bandwidth consumption.

INTERVAL TEMPORAL LOGIC

Interval temporal logic (ITL) is providing s solid mathematical foundation for processing a sequence of state. We are using concepts from ITL to validate the event processing model we are using in our middleware.

The key notion of ITL (H Janicke, Cau, Siewe, & Zedan, 2012) is an `interval`. An interval σ is considered to be a (in)finite sequence of states $\sigma_0 , \sigma_1 . . . ,$ where a state σ_i is a mapping from the set of variables `Var` to the set of values `Val`. The length $|\sigma|$ of an interval $\sigma_0 . . . \sigma_n$ is equal to n (one less than the number of states in the interval, so a one state interval has length 0).

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

Figure 2: Syntax of ITL

The syntax of ITL is defined 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 [4], however uses the projection operator f1Δf2 as primitive and derives the operator f* as introduced in (Moszkowski, 1995).

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

- **skip**: unit interval (length 1, i.e., an interval of two states).
- **f1;f2**: ("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. This is illustrated in *Figure 3*.
- **f1Δf2**: ("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.

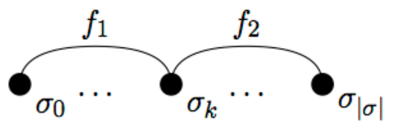


Figure 3: Informal Semantics of f1;f2

An example is depicted in *Figure 4*.

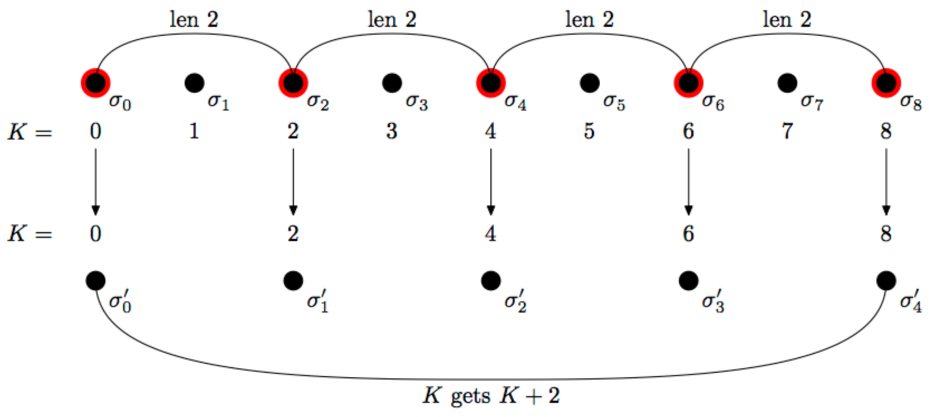


Figure 4: Example of Temporal Projection

In the interval σ the value of K increases from 0 to 8 in steps of one. The interval σ satisfies `(len(2))△(K gets K + 2).` `(len(2))` is true if the interval is of length two and `(K 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 `len(2).` The formal definition of this operator is given in (Moszkowski, 1995).

- ○`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.

The following lists some of the derived constructs used in the remainder of this paper. The binary operators ∨ (or) and ⊃ (implication) are derived as usual:

- ○`f = skip;f` (read "next f"), means that f holds from the next state. Example: ○ `(X = 1):` Any interval such that the value of X in the second state is 1 and the length of that interval is at least 1

- `more =` ○`true` means the non-empty interval, i.e., any interval of length at least one.

- `empty =` ¬`more` means the empty interval, i.e., any interval of length zero (just one state).

- `inf = true;false` means the infinite interval, i.e., any interval of infinite length.

- `finite = ¬inf` means the finite interval, i.e., any interval of finite length.

- `◇f = finite;f` (read "sometimes f"), i.e., any interval such that f holds over a suffix of that interval. Example: $\Diamond X \neq 1$: Any interval such that there exists a state in which X is not equal to 1.

- `□f = ¬ ◇¬ f` (read "always f"), i.e., any interval such that f holds for all suffixes of that interval. Example: $\Box$ (X = 1): Any interval such that the value of X is equal to 1 in all states of that interval.

- `fin f= □(empty⊃f)` defines the final state, i.e., any interval such that f holds in the final state of that interval.

- `halt f = □(empty = f)` terminate the interval when f holds.

- $\exists v \cdot f = \neg\forall v \cdot \neg f$ existential quantification.

- $len(e) = \begin{cases} false & if\ e\ < 0 \\ empty & if\ e = 0 \\ skip; len(e-1) & if\ e > 0 \end{cases}$ holds if the interval length is e.

- $v\ gets\ e = \Box(more\ \supset (\bigcirc v) = e)$ gets, i.e., in every state except the initial state the variable v will be assigned the value of e evaluated in the previous state.

- $f^* = f\ \Delta true$ (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.

Complex Event Processing

Complex event processing (CEP (Luckham, 2002)) can be seen as continuous and incremental processing of event streams from multiple sources based on declarative query and pattern specifications with near-zero latency.
In our work we consider sensor readings as events. Thus, we can see them as a continuous stream of events.
An event is a basic concept of CEP. It defines something that happened.
There are plenty of events in our every-day life, for example: a historical or social happening, a car accident or receiving of a text message. There are different formal definitions of this term in the literature (Gehani, Jagadish, &

Shmueli, 1993; Sayal, 2004). In this text, we will use the term *event* as any happening of interest that can be observed, recorded and reacted on in a system.

An event is created in two distinct steps: *observation* and *adaptation*. Firstly, a particular activity of a system should be observed without changes to the system's behaviour. Then observed activities are transformed into event objects that can be processed by the CEP system. The transformation is usually done by special entities called *adapters*.

In natural language or everyday life an event is defined as something that happens or is supposed to happen. This understanding is partly valid for events in CEP. Because in CEP an event is represented which does an object trigger by an activity or happening, the significance, but it is not the activity itself.

An event is also not just a pure message. The form of such an event could be represented by a message but it also needs to cover significance and relativity. Dependencies and relationships between events can have different form and are often the main points of interest for CEP applications

There are three common relationships between events (Sethi, 2001). They are:

1. *Time*. Time is used to order events, defining which of the two events *A* or *B* happened earlier. Usually, as soon as an event is created, the current time is added to the event object in a form of a timestamp. So, timestamps define time relationships between events. This type of relationship depends on the clocks in the system. Events can have different time relationships corresponding to different clocks. Events' comparability in this case depends upon whether the clocks are synchronized.
1. *Cause*. Event *A* causes event *B*, if the activity signifying event *A* had to happen in order for activity *B* to happen. Accordingly, causality is a relationship of dependency between activities. If event *A* can happen only after event *B*, then event *B* depends on event *A*, or event *A* caused event *B*. When neither of the events depends on each other, we say the events are independent. The relationship of time and cause is expressed in the following axiom, which is valid for most systems: If event *B* is caused by event *A* in system *S*, than no clock in system *S* will give an earlier timestamp to event *B* than to event *A*. Consequently, if the clocks can observe two dependent events, they will always observe them in the same order.
2. *Aggregation*. If the activity corresponding to event *A* consists of other activities corresponding to a set of events $B_1 \ldots B_n$, then event *A* is an aggregation of the events B_i. In this case we say that events $B_1 \ldots B_n$ are members of event *A*. Aggregation is a mean to abstract events, enabling introduction of vertical dependency. Furthermore, the activity signified by event *A* is more complex than the activities

of member events. Because of that, its signifying event is called a complex event. It is easy to see, that aggregation leads to a causal relationship between the complex event and its members.
Activities signified by complex events often happen over a period of time and instead of a timestamp a time interval is used. The time interval starts with the earliest activity and ends with the end of the latest activity from the event's members.

Actor Model

Handling big data in a fast and efficient way might require to process data in a concurrent, parallel way. The actor model has its theoretical roots in concurrency modeling (Hewitt, Bishop, & Steiger, 1973) and message passing concepts. The fundamental idea of the actor model is to use actors as concurrent primitives that can act upon receiving messages in different ways:

- Send a finite number of messages to other actors.
- Spawn a finite number of new actors.
- Change its own internal behavior, taking effect when the next incoming message is handled.

For communication, the actor model uses asynchronous message passing. In particular, it does not use any intermediate entities such as channels. Instead, each actor possesses a mailbox and can be addressed. These addresses are not to be confused with identities, and each actor can have no, one or multiple addresses. When an actor sends a message, it must know the address of the recipient. In addition, actors are allowed to send messages to themselves, which they will receive and handle later in a future step.
Here, the actor model will help to separate processing units in a distributed setup. Instead of passing only messages from actors to actors, the middleware is also passing expression – the core logic to process data – to actors. Means, the policy obligations can be seen as injected behavior of an actor. Thus, the actor model helps to provide a foundation for the distributed hyper-scale setup.

BASIC CONCEPTS

In this section we are providing an overview of used core concepts. In particular, we will introduce time dependent offerings that need to be processed. We are also introducing a concept (policies) based on event-condition-action paradigm so that we can express rules over continuous stream of data to trigger actions. Therefore, we are defining a data stream model and combine it with the notion of time. Finally, we explain how these policies can be validated against ITL logic.

Time Dependent Service Offerings

Properties of services are considered to be non-functional or functional. Such properties are used for service selection or context-based service discovery. The available approaches are based on the fact that properties are pulled from service repositories (that is from service metadata) or possibly from the services directly before the algorithm determine the most relevant service for a given context. Repositories are useful for static data and polling services directly works if a small number of properties of a small number of services is of interest. We believe that there is an emergent need to provide methods to enable the continuous evaluation of functional and non-functional properties especially in the case where the number of services is high. We define static properties `ps` as constant over time, such as a location of a printer, the vendor of a printing machine, or the number of a taxi etc. Temporal properties `pt` are changing over time. Using these, we define non-functional properties `NFP` as a tuple of static properties and dynamic properties:

$$\mathtt{NFP(t)} = \langle \mathtt{ps,pt(t)} \rangle$$

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

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

This presents the static data schema; like a snapshot in time. Temporal aspects are covered by events and therefore we would see different data at different points in time.

Event Policies

Policies refer to obligations placed on a service 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 modeled similar to the well-understood Event-Condition-Action paradigm (Twidle, Lupu, Dulay, & Sloman, 2008; Uszok et al., 2003). 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 (H Janicke et al., 2012; Helge Janicke, Cau, Siewe, Zedan, & Jones, 2006), is that the condition bridges between two events, thus does not require the storage of large amounts of historical data.

Informally a policy *pol* is a set of rules of the following structure:

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

The `<Source>` 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 events (e.g. a GPS-Update) that are domain dependent and 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 that is described in the next section.

Event Policy Validation with ITL

Evaluating the policy pols of the service s against this interval is a two-stage process.

Stage 1

First, for every rule $r \in pols$ 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**.

Stage 2

Second, 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:

$$\sigma_s \vDash ohalt(evt_r)\Delta(cnd_r\Delta\Box act_r)$$

Here `halt(evtr)Δf` 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 rationale for separating the two steps is that the filtering of event streams based on simple events (evt_r) can be implemented very efficiently, whereas the complexity of the evaluation of the conditions cnd_r is more complex and can in certain cases grow linearly with the number of states that are bridged. Thus the initial reduction using the event filter reduces the complexity of the latter evaluation. The overall service specification is then constructed from this as:

$$\sigma_s \vDash \bigwedge_{r\epsilon\, pol_s} ohalt(evt_r)\Delta(cnd_r\Delta\Box act_r)$$

The specification of act_r is not detailed here and we only consider that the relevant action is initiated in that state of the service interval. The model can be implemented straightforwardly from its semantics using some functional programming, resulting in the following code:

```
//run
// Create the source
let evtmodel =
    [|(1,1,0);(1,0,1);(1,1,0);
    (1,0,1);(0,0,0);(0,0,1);
    (1,0,0);(1,0,0)|]
let s = new Source()
// example rule evaluation
s.AsStream // selecting events Evts[0]
|> Stream.filter( fun e -> e.p == (1,_,_))
// show selected events, testing only
> Stream.print ("State %d: Evts[0] = 1", e.state)
// select every second event only
|> Stream.len 2
// show selected events, testing only
|> Action (fun e ->
printfn "State %d: Action on every 2nd Evts[0]"
(e.Item(1).state))
|> ignore
// create test event trace for the service
evtmodel |> skip s
```

Here three events are modelled for the service, and an example trace is generated by the defined `evtmodel`. 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 filter condition (*Stream.filter(fun e -> e.p == (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 (*Stream.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 will produce the following output

```
State 0: Evts[0] = 1
State 1: Evts[0] = 1
State 1: Action on every 2nd Evts[0].
State 2: Evts[0] = 1
State 3: Evts[0] = 1
State 3: Action on every 2nd Evts[0].
State 6: Evts[0] = 1
State 7: Evts[0] = 1
State 7: Action on every 2nd Evts[0].
```

The event *Evts[0]* is raised in the states 0, 1, 2, 3, 6 and 7 as also indicated by the control outputs. The Action is triggered on every second occurrence of the event, namely in states 1, 3 and 7.

ARCHITECTURE

As we have pointed out above to achieve almost zero latency data processing, data must be available at the place where the consumer needs it. So, instead of pulling data at request time from data sources, data should be pushed to a data provider. If we apply a scan-based approach to an SOA this would mean that a consumer is pulling data from services (see *Figure 5*a). In contrast and event-based approach would mean that services are pushing data to a consumer (see *Figure 5*b).

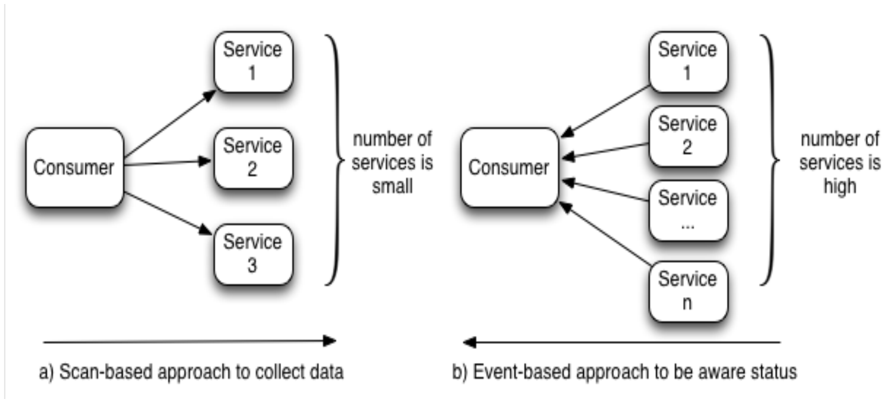

Service 1
Service 2
Service 3
Consumer
number of services is small
a) Scan-based approach to collect data
Service 1
Service 2
Service ...
Service n
Consumer
number of services is high
b) Event-based approach to be aware status

Figure 5: Metaphor comparison of request-response and event-based in SOA

However, using event-based models 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 a data provider (e.g. the selector) there is a vast amount of overhead by unnecessarily transferring data – a waste of bandwidth. We introduced *event policies* on the data source so that we can control when and which data is pushed to the consumer. These *policies* 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 – should enable low capacity filtering by being context-aware.

Conceptual Architecture

As a central instance we use a Mediator (see *Figure 6*). 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. Then, an incoming consumer request is handled as a query on subscribed service properties. Instead of pulling at request time all the data from all services 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.

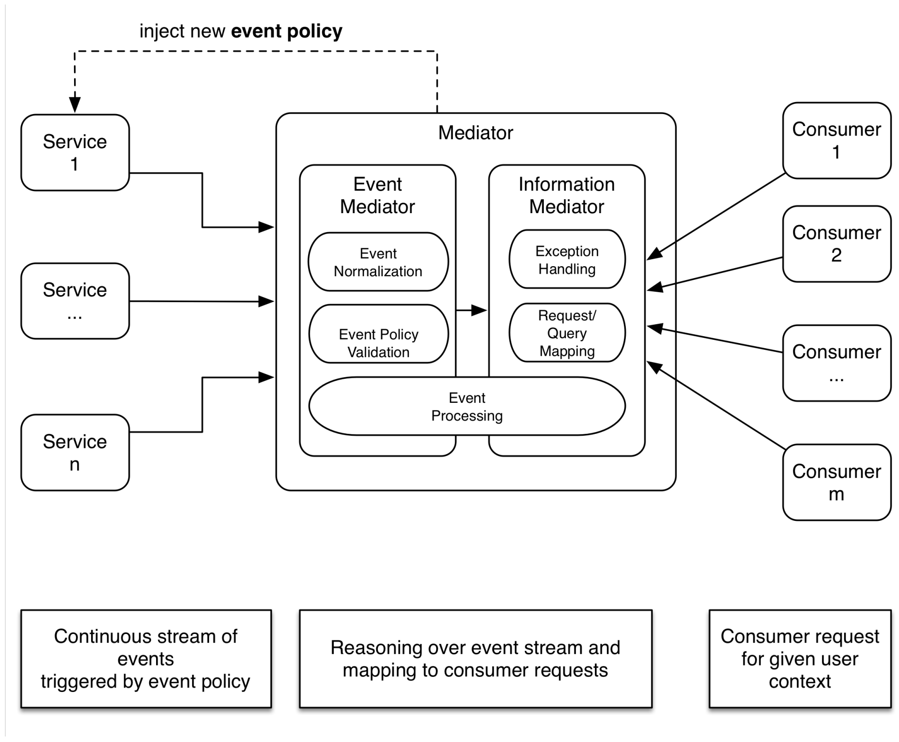


Figure 6: Logical mediation service architecture

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

Event Mediator

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 *Event Mediator* maintains a mapping table to transform incoming events from endpoints into a normalized data stream. Let's assume the *Service1* provides events containing temperature in Celsius while *Service2* provides the temperature data in Fahrenheit. The *Event Mediator* normalizes event streams internally before the event data is forwarded 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. Here 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.

Information Mediator

The *Information Mediator* maps consumer request to queries on continuous event streams provided by the *Event Mediator*. On the consumer side the framework still offers a normal Web Service interface, which internally needs to be transformed into a query, which is 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.

MIDDLEWARE

In this section we will show how event processing within the mediator and event polices can be expressed Since a stream *S* is considered as an (in)finite sequence of events e_i. In general streams can be filtered, mapped, or zipped (joined). While each of these operations produces a new output stream operators can be piped (|>). In pseudo-code filtering for odd numbers and multiplying them with 2, could look like this:

```
source
|> filter(x -> isOdd(x))
|> map (x -> x *2)
|> action
```

Joining two streams will look like this:

```
source1 |> zip(source2) |> action
```

This is exactly what we have explained in basic concepts with event policies, which have the form of

```
source |> condition |> action
```

In our middleware there are the following core elements (see *Figure 7*) as described in basic concepts as an ECA policy:

- **Source (src)** produces or injects a continuous stream of events. There are several implementations of a source. A simple one is the *RandomSource*, which creates random values by a given interval. The *PushSource* exposes an endpoint to which events can be pushed form outside or actions. A *PullSource* can be used to get data in a request-

response way. The sources enable the convergence of pull and push systems.
- **Condition** (**cnd**) defines the transformation of events as a pipeline with operators like filter, map, zip, ….
- **Action (act)** triggered by the condition result. An action can either simply sends out the event to other sources can trigger an action or simply output the result. The action is also used to hold the final state for a request.

In addition to these core elements we define the following additional concepts:

- **Expression** is defined by source, condition, action `expr = do: source |> condition |> action`
- **ReActor** is lightweight container for expressions. A ReActor is an extended actor since the actor behavior is injected `ReActor.send(reactor, expr)`. Please note that a ReActor is not a component that is reacting to things, so read "Re-Actor" and not "reactor"). A ReActor instance is defined by its unique process identifier (PID).
- **Node** is process hosting multiple ReActors. A node can be identified by its unique node identifier (NID).

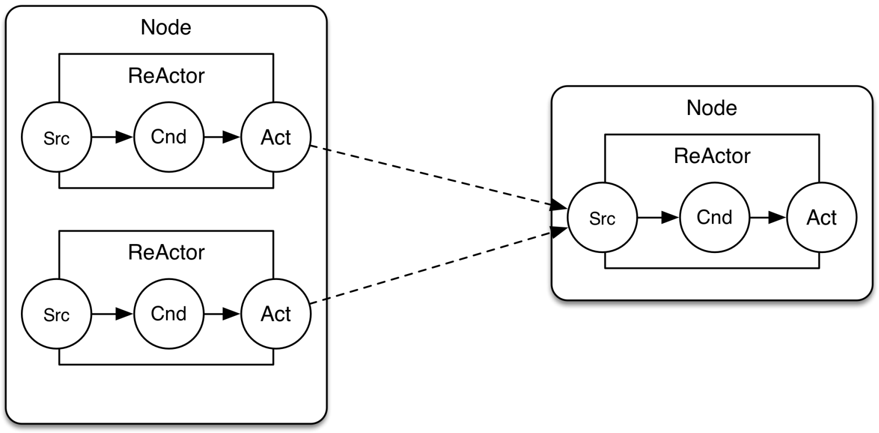


Figure 7: Terminology of ReActor middleware

ReActors can be connected with sources and actions. By connecting *ReActors* it is possible to build highly distributed and scalable processing pipelines. Here we are benefiting from the actor approach. Compared to classical actor a ReActor is not only encapsulating his state and can receive events, it is possible to inject the behavior of an actor. This is an extension to the original actor model.

We can find similar approaches in Cloud Haskell (Epstein, Black, & Peyton-Jones, 2012) where closures can be serialized to a mailbox as expressions or functions. This is more restricted to closures and is not as complete as our approach. In Orleans (Bykov et al., 2011) we can find a actor model implementation on C#/.Net but the flexibility is limited to sending around behavior in form of expressions.

The notion of sources and actions allow us to have a flexible integration of push-based and pull-based approaches. With *PullSources* classical REST services can be integrated (see *Figure 8*). Another novelty is the lightweight nature of a *ReActor*. While other systems, like Yarn, are realizing this as processes, a *ReActor* is simply an actor. With having a supervisor it is possible to even build a reliable system.

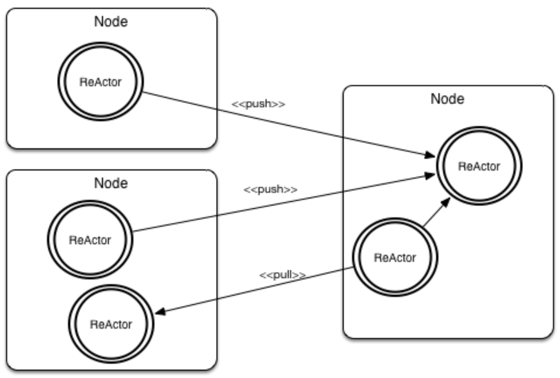


Figure 8: ReActor topology

Implementation

The middleware is ideally realized on top of a functional programming language, such as Haskell (https://www.haskell.org), F# () or Erlang (Armstrong, 1997, 2010; Larson, 2009) because expressing ITL logic in a functional programming language is straight forward..

Haskell is the purest functional programming language and would be ideal to translate ITL logic into functions. But Haskell has limitations in terms of actor model and distribution. As mentioned the Cloud Haskell project is trying to build an actor comparable framework but the project is not mature enough compared to F# or Erlang regarding this aspect.

F# is not a pure functional programming language. It is a mixture of OOP concept and functional elements. Expression ITL logic as functions is feasible and there are several actor model implementations for F#, like AKKA, but serialization of expressions is quite difficult.

The actor model in Erlang is providing a great foundation for the hyper-scaling setup. It can be seen a first class citizen within the Erlang framework. In fact, Elixir (http://elixir-lang.org) is used for Reactor. Elixir is an extension on top of Erlang. In Elixir everything is an expression and can be serialized easily to other instances. This build-in functionality in combination with the superior actor model implementation makes Elixir the first choice for realizing our ReActor middleware.

Consider Figure 9, which shows the taxi scenario in a simplified version. The logic to find the nearest taxi is described as a set of expressions considering the topology. Here it is simple since we do have only on server and multiple taxis. The expression on the taxi is sending an update event to the server only if the taxi has changed its position by more than 50 meters and is available. The *ReActor* on the server is collecting the taxi events and keeps the state. Whenever a customer is asking for a taxi providing his GEOlocation, the Reactor has to find the taxi at that rime with minimal distance to the customer.

To enable a processing pipeline end-2-end follows these steps:

Step 0: Client node (NID#1) registers itself in registry

Step 1: Author the expression

Step 2: By calling `Node.start(…)` expression will be send to the node with NID#2

Step 3: The node with `NID#2` will start a new ReActor instance hosting the given expression (expr1)

Step 4: The ReActor will register itself at the *Registry*. In this case with the tag “GetNearestTaxi” which needs to be defined during expression definition.

Step 5: Next expression will be send to node with NID#1

Step 6: The node with NID#1 starts a *ReActor* with given expression (expr2). This can be scaled out to multiple nodes, such as taxis in our example.

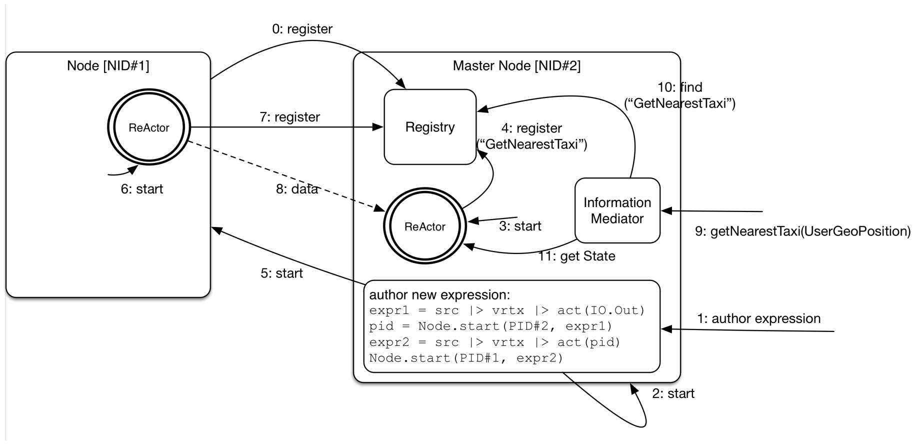
Step 7: The new *ReActor* will register itself at the *Registry*

Step 8: Finally, the Reactor in NID#1 sends data to *ReActor* in NID#2

Step 9: Let’s assume there is a customer looking for the nearest taxi. He has to provide his GEOLocation and calls the *Information Mediator* endpoint.

Step 10: The *Information Mediator* finds the *ReActor* with tag “GetNearestTaxi” in registry

Step 11: Without any delay the *Information Mediator* collects state from *ReActor* and sends this back to customer.



```
author new expression:
expr1 = src |> vrtx |> act(IO.Out)
pid = Node.start(PID#2, expr1)
expr2 = src |> vrtx |> act(pid)
Node.start(PID#1, expr2)
```

Figure 9: ReActor usage

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
2. a requirement to reduce the amount of data transfer (recall that this was large because of the amount of small messages, not because the data in itself being intrinsically large).

The experimental 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 occurring during a one second time interval in the push model and combined pull-push model. The overall setup considered setting with up to 60000 data sources.

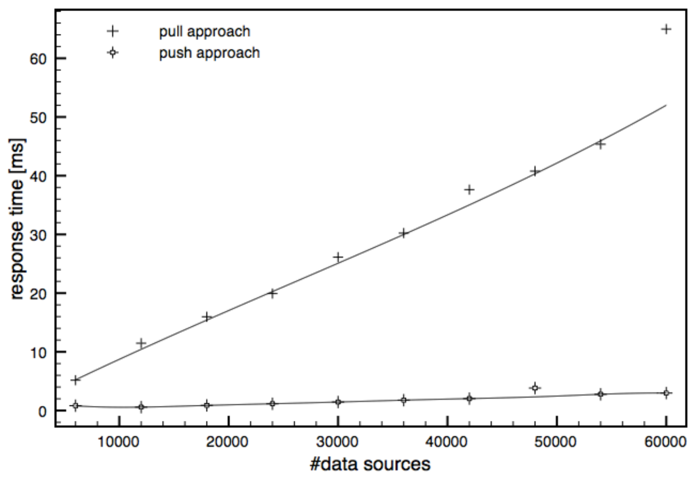


Figure 10: Latency of scan-based approach (pull) vs. event-based approach (push)

Testing pull and push approaches is quite complex since there is a big number of data sources required to get some significant results. Nonetheless, we had to distinguish between pure latency as a result of the different concepts and latency caused by physical setup, such as network latency and latency used by semantic processing. For testing we decided to run everything on one machine and to simulate each data sources as a small object in our test setup. This object can be seen as an atom, a logical, self-contained unit. By having all simulated data sources in memory on one machine we can say that latency caused by the physical setup can be neglected. The simulated objects (data sources) were simply holding a random value and the query in the setup was to find the object whose value is the closest to a given number. This simple setup matches the taxi example where the value would represent the geo position of the taxi and the given number would be the customer geo position, but also reflects the typical scenario where we see the approach apply. In the pull approach we are iterating over all objects (simulated data sources) 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, such as

```
values
|> filter (x -> (x-number))
|> (take 10).Min()
|> out
```

The results are presented in *Figure 10*. While the latency is increasing linearly for the pull approach, it remains almost constant for the push approach. We also find that all values for the push approach are far lower than those for the pull approach, with for example approx. 4ms vs. 65ms for 60000 data sources. This clearly indicates that the push approach is superior compared to the pull approach in terms of latency and the trends show that for a growing number of data sources as those expected in scenarios like the Internet of Things is delivering performance close to no latency. We also want to point out that this is the pure measured latency ignoring network and processing delays – once these are added as additional factors to the evaluation, latency will 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 measured in the isolated setting).

The drawback of the push approach is the number of data items send from the data sources to the mediator. Therefore, 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 pure push approach compared with the number of messages which sent in a push approach with filtering. The latter is expected send fewer messages because of the injected policy. For testing we extended the objects with another random value representing a state (on or off). The status here is a random number, either 0 or 1. The rule is saying that the objet should push only messages when the state is 1. *Figure 11* shows the results as we expected. With a simple rule we can roughly save 50 percent of exchanged messages (based on the randomly changing values). It is clear that with even more specific rules and real data the number of messages can be further reduced.

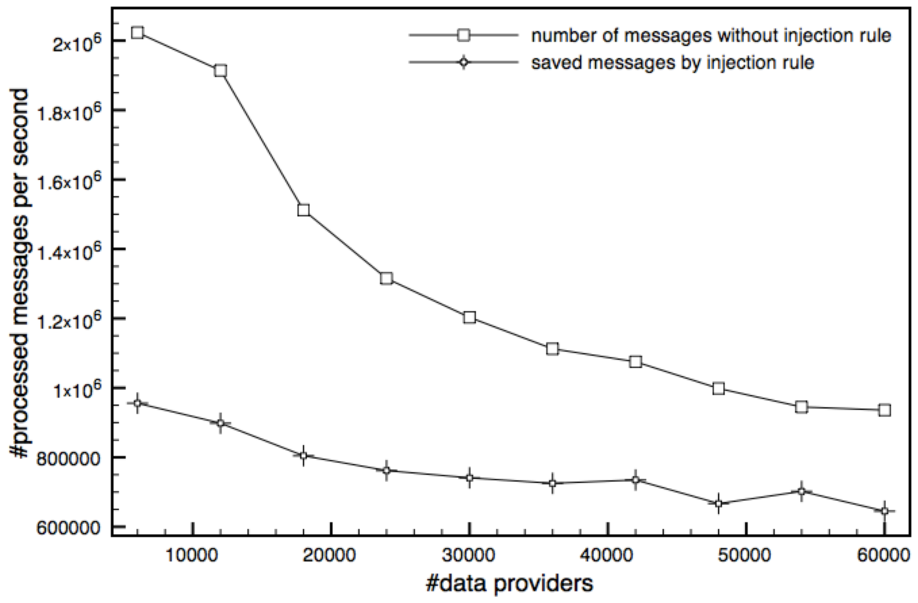


Figure 11: Event-based approach with injected rules and without

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

CONCLUSION

We presented a new approach, which combines event processing based on interval temporal logic for fast data processing for SOA and event-based systems. Our approach investigates processing of service offerings with a huge number of potential services or data sources. 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 event policies 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 behavior.

By using the ReActor approach we are providing a concept which is not limited to any SOA base approaches. The usage of the sources concept enables to integrate in any existing system. Requirements like scaling and distribution are covered by the extension of the actor model. This enables to avoid bottlenecks and single point of failures. Via ReActor we can easily scale to a number of actors, which is required.

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. In future work we will also further validate and investigate limitations of our approach.

REFERENCES

Abadi, D., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Erwin, C., … M. (2003). Aurora: a data stream management system. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data* (p. 666). New York, NY, USA: ACM. doi:http://doi.acm.org/10.1145/872757.872855

Armstrong, J. (1997). The development of Erlang. *ACM SIGPLAN Notices*.

Armstrong, J. (2010). Erlang. *Communications of the ACM*.

Bonifati, A., Ceri, S., & Paraboschi, S. (2002). Pushing reactive services to XML repositories using active rules. *Computer Networks*, *39*, 645–660.

Bykov, S., Geller, A., Kliot, G., Larus, J. R., Pandya, R., & Thelin, J. (2011). Orleans : Cloud Computing for Everyone. *Proceedings of the 2nd ACM Symposium on Cloud Computing SOCC 11*, 1–14. Retrieved from http://dl.acm.org/citation.cfm?id=2038916.2038932

Cau, A., Moszkowski, B., & Zedan, H. (2011). *The ITL homepage: http://www.cse.dmu.ac.uk/STRL/ITL*. Retrieved from http://www.cse.dmu.ac.uk/STRL/ITL

Chakravarthy, S., & Mishra, D. (1994). Snoop: an expressive event specification language for active databases. *Data Knowl. Eng.*, *14*(1), 1–26. doi:10.1016/0169-023X(94)90006-X

Demers, A., Gehrke, J., Hong, M., Riedewald, M., & White, W. (2006). Towards expressive publish/subscribe systems. *Advances in Database Technology-EDBT 2006*, 627–644. Retrieved from http://www.springerlink.com/index/y684305339173080.pdf

Duan, Z.-H., & Koutny, M. (2004). A framed temporal logic programming language. *J. Comput. Sci. Technol.*, *19*(3), 341–351. doi:10.1007/BF02944904

Edge, M., Sampaio, P., Philpott, O., & Choudhary, M. (2008). A Policy Distribution Service for Proactive Fraud Management over Financial Data Streams. *Services Computing, IEEE International Conference on*, *2*, 31–38. doi:http://doi.ieeecomputersociety.org/10.1109/SCC.2008.105

Epstein, J., Black, A. P., & Peyton-Jones, S. (2012). Towards Haskell in the cloud. *ACM SIGPLAN Notices*.

Gehani, N. H., Jagadish, H. V, & Shmueli, O. (1993). COMPOSE: A System For Composite Specification And Detection. *Lecture Notes In Computer Science Vol 759*, 3–15. Retrieved from http://portal.acm.org/citation.cfm?id=725348

Hewitt, C., Bishop, P., & Steiger, R. (1973). A Universal Modular ACTOR Formalism for Artificial Intelligence. In *IJCAI* (pp. 235–245).

Janicke, H., Cau, A., Siewe, F., & Zedan, H. (2007). Deriving Enforcement Mechanisms from Policies. In *Proceedings of the 8th IEEE international Workshop on Policies for Distributed Systems (POLICY2007)* (pp. 161–170).

Janicke, H., Cau, A., Siewe, F., & Zedan, H. (2012). Dynamic Access Control Policies: Specification and Verification. *The Computer Journal*. doi:10.1093/comjnl/bxs102

Janicke, H., Cau, A., Siewe, F., Zedan, H., & Jones, K. (2006). A Compositional Event & Time-based Policy Model. In *Proceedings of POLICY2006, London, Ontario, Canada* (pp. 173–182). London, Ontario Canada: IEEE Computer Society.

Larson, J. (2009). Erlang for concurrent programming. *Communications of the ACM*.

Luckham, D. (2002). *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Amsterdam: Addison-Wesley Longman. Retrieved from http://www.amazon.com/Power-Events-Introduction-Processing-Distributed/dp/0201727897

Marganiec, S. R., Tilly, M., & Janicke, H. (2014). Low-Latency Service Data Aggregation Using Policy Obligations. *2014 IEEE International Conference on Web Services*, 526–533. doi:10.1109/ICWS.2014.80

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

Reiff-Marganiec, S., Yu, H., & Tilly, M. (2009). Service selection based on non-functional properties. In *Service-Oriented Computing-ICSOC 2007 Workshops* (Vol. 4907, pp. 128–138). Springer. Retrieved from http://www.springerlink.com/index/h510k782167228r8.pdf

Roitman, H., Gal, A., & Raschid, L. (2009). Web Monitoring 2.0: Crossing Streams to Satisfy Complex Data Needs. In *Proceedings of the 2009 IEEE International Conference on Data Engineering* (pp. 1215–1218). IEEE Computer Society. Retrieved from http://ie.technion.ac.il/~avigal/cosmos.pdf

Sayal, M. (2004). Detecting time correlations in time-series data streams. *Hewlett-Packard Company*. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.58.3219&amp;rep=rep1&amp;type=pdf

Sethi, A. S. (2001). SEL, a new event pattern specification language for event correlation. *Proceedings Tenth International Conference on Computer Communications and Networks Cat No01EX495*, *00*(C), 586–589. doi:10.1109/ICCCN.2001.956327

Tian, C., & Duan, Z. (2009). Complexity of propositional projection temporal logic with star. *Mathematical. Structures in Comp. Sci.*, *19*(1), 73–100. doi:10.1017/S096012950800738X

Tilly, M., & Reiff-Marganiec, S. (2011). Matching customer requests to service offerings in real-time. In *Proceedings of the 2011 ACM Symposium on Applied Computing - SAC '11* (p. 456). New York, New York, USA: ACM Press. doi:10.1145/1982185.1982285

Tu, Y. C. C., Liu, S., Prabhakar, S., & Yao, B. (2006). Load shedding in stream databases: a control-based approach. In *Proceedings of the 32nd international conference on Very large data bases* (pp. 787–798). VLDB Endowment. Retrieved from http://portal.acm.org/citation.cfm?id=1164195

Twidle, K., Lupu, E., Dulay, N., & Sloman, M. (2008). Ponder2 - A Policy Environment for Autonomous Pervasive Systems. In *Policies for Distributed Systems and Networks, 2008. POLICY 2008. IEEE Workshop on* (pp. 245–246). doi:10.1109/POLICY.2008.10

Uszok, A., Bradshaw, J., Jeffers, R., Suri, N., Hayes, P., Breedy, M., … Lott, J. (2003). KAoS: policy and domain services: toward a description-logic approach to policy representation, deconfliction, and enforcement. In *Proceedings POLICY 2003 Policies for Distributed Systems and Networks* (pp. 93–96).

Yu, H. Q. Q., & Reiff-Marganiec, S. (2008). 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. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.142.3654&amp;rep=rep1&amp;type=pdf

Yu, T., Zhang, Y., & Lin, K.-J. (2007). Efficient algorithms for Web services selection with end-to-end QoS constraints. *ACM Transactions on the Web*, *1*(1), 6–es. doi:10.1145/1232722.1232728