Managing Expectations: Runtime Negotiation of Information Quality Requirements in Event-based Systems

Sebastian Frischbier[1], Peter Pietzuch[2], and Alejandro Buchmann[1]

[1] TU Darmstadt
{frischbier, buchmann}@dvs.tu-darmstadt.de
[2] Imperial College London
p.pietzuch@imperial.ac.uk

Abstract. Interconnected smart devices in the Internet of Things (IoT) provide fine-granular data about real-world events, leveraged by service-based systems using the paradigm of event-based systems (EBS) for invocation. Depending on the capabilities and state of the system, the information propagated in EBS differs in content but also in properties like precision, rate and freshness. At runtime, consumers have different dynamic requirements about those properties that constitute quality of information (QoI) for them. Current approaches to support quality-related requirements in EBS are either domain-specific or limited in terms of expressiveness, flexibility and scope as they do not allow participants to adapt their behavior. We introduce the generic concept of **expectations** to express, negotiate and enforce arbitrary requirements about information quality in EBS at runtime. In this paper, we present the model of expectations, capabilities and feedback based on generic properties. Participants express requirements and define individual tradeoffs between them as expectations while system features are expressed as capabilities. We discuss the algorithms to (i) negotiate requirements at runtime in the middleware by matching expectations to capabilities and (ii) adapt participants as well as the middleware. We illustrate the architecture for runtime-support in industry-strength systems by describing prototypes implemented within a centralized and a decentralized EBS.

Keywords: event-based systems, quality of information, self-adaptive systems, runtime negotiation, malleability

1 Motivation

Having information of adequate quality available at the right time in the right place is vital for software systems to react to situations or support decisions. Supply chain management based on the Internet of Things (IoT) and data centre monitoring are just two examples of reactive systems where information provided by data sources has to be interpreted and where false alarms, missed events or otherwise information of inadequate quality carries a cost [18]. Event-based systems (EBS) and service-oriented architectures (SOA) complement each other

well to leverage those streams of dynamic real-time information in enterprise software systems and react on meaningful events in a timely manner: software components can be exposed as services for direct communication while also acting as participants of an EBS to follow an indirect communication model [8,10]. EBS are anonymous, information-centric systems with many-to-many communication: loosely-coupled software components (publishers) publish notifications about events they are confident to have detected (e.g., critical workload at node) as messages which are pushed to interested components (subscribers) by a middleware. Information exchanged in EBS is characterized by type (e.g., `temperatureEvent`), content (e.g., *temperatureCelsius*=50) and quality-related properties (e.g., rate of publication, confidence, precision, trustworthiness) [14].

Subscribers in EBS require information with sufficient quality of information (QoI) to decide whether to react or not: being notified too late or causelessly due to false positive can have severe consequences [18]. Whether some QoI is sufficient depends on the information's properties fitting the purpose it is intended to be used for; this is application-specific and dynamic as it depends on the context of each subscriber [6]. For example, monitoring data about a virtual machine delivered at a given rate and confidence might be (i) sufficient for the purpose of one subscriber while another subscriber might need the same type of data at a higher rate but would tolerate less confidence; (ii) sufficient for a subscriber as long as there is no indication of malfunction at the monitored entity - in case of anomalies the same data is required at high rate for root cause analysis [18].

QoI in EBS depends on the system satisfying individual requirements about quality-related properties at runtime [4]. Subscribers have to be able to (i) define requirements about arbitrary quality-related properties of information they want to consume; (ii) expose individual tradeoffs between those requirements if they are willing to accept degradations in exchange for getting other requirements satisfied; (iii) adapt requirements at runtime to reflect changes to their current situation; and (iv) get feedback about the state of their requirements to decide if their needs are satisfied or if they have to adapt. At the same time, delivering information with specific properties comes at a cost for the system as it depends on the current configuration of available publishers and the middleware [3]. Thus, the system has to decide at runtime *how* to satisfy *which* requirements.

Runtime support for quality-related properties in EBS is currently limited in terms of expressivity, extensibility and scope; feedback is not provided at runtime [2,11]. Requirements about quality-related properties in EBS can be implicitly supported by publishers either by using types that encode quality-related properties in their name (e.g., CpuUsage_rate50_confidence70), or by adding metadata to the content of each published message (e.g., *rate=50*, *confidence=70*). Subscribers can express their requirements by subscribing to the type they are interested in using the common API of EBS [22]. However, this restricts the set of available properties to those determinable by publishers at design-time, excluding important runtime properties like latency and reliability that are provided by the middleware. For encoded types, it would also result in an unmanageable growth of available types for different combinations of

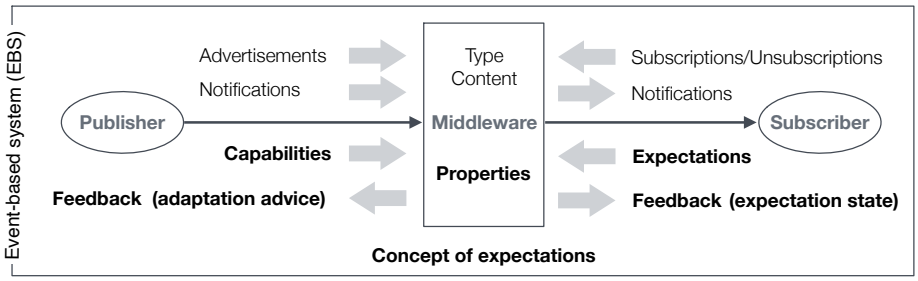

Fig. 1: Our concept extends the model of EBS (top) with capabilities, expectations and bidirectional feedback for runtime adaptation (bottom, bold).

quality-properties, as well as, traffic overhead as the same information has to be processed for multiple encoded types [9]. Systems providing explicit support for quality-related properties like IndiQoS [9], Adamant [15] or Harmony [28] focus on a fixed set of middleware-related properties at a low level of abstraction. They try to satisfy requirements by adapting the middleware on the transport protocol level and do not enforce requirements about properties that would require publishers to adapt at runtime, limiting the scope of runtime flexibility.

In this paper, we propose the concept of **expectations** as a generic approach to support QoI in EBS as a first-class citizen and enable participants to adapt at runtime. Fig. 1 shows how our approach complements the paradigm of EBS.

Our key idea is to define quality-related properties like *rate*, *confidence* or *latency* in a generic way together with actions that define how those properties can be adjusted at runtime. Requirements (*expectations*) and the system state (*capabilities*) defined as ranges over such properties can be efficiently matched in the middleware at runtime to identify the extent to which the system would have to adapt to satisfy requirements. Based on this assessment, requirements can be declined or satisfied by adapting the system using platform-specific instantiations of the associated actions. Subscribers receive feedback about the state of their requirements while publishers get feedback about the usage of their capabilities, including advice to adapt if necessary.

For example, application M has to monitor the temperature of a chemical process during manufacturing to detect anomalies and trigger a dedicated workflow. In terms of QoI, M subscribes to notifications for `temperatureEvent` with an expectation about *rate* and *confidence*: it requires notifications to be of 75-95% confidence (minimizing false-positives/negatives) while they could be published at a low rate of 5-10 events/minute. A temperature sensor P, currently publishing 2 events/minute with 90% confidence, is able to publish up to 60 events/minute with a maximum confidence of 80%, expressing this as capabilities for *rate* and *confidence*. Matching M's expectation to available capabilities, the middleware realizes that P is suitable but has to adapt, advising it to do so.

Subscribers can express requirements and individual tradeoffs between them as expectations in a consistent and information-centric way over arbitrary prop-

erties. Publishers expose their general capabilities as well as the state they are currently operating at to brokers as capabilities. Support for new properties can be realized by extending the set of available properties, their relationships and by associating suitable actions for manipulating them at the middleware.

Expressing expectations and capabilities as ranges of accepted and provided values over properties automates the process of runtime negotiation: matching requirements to the system state is reduced to a range matching problem between corresponding properties. Furthermore, requirements become *malleable* due to the individual tradeoffs defined by subscribers, giving the system more degrees of freedom when deciding on the extent of adaptation necessary. Feedback enables participants to adapt their behavior at runtime and extends the scope of supported properties to those influenced by publishers.

The concept of expectations complementary extends the paradigm of EBS without compromising the model of indirect many-to-many communication, making it backward compatible. As shown in Fig. 1, expectations and capabilities can be defined independently of advertisements, notifications or subscriptions. They are matched only at the middleware, preserving the anonymity of the associated participants necessary for scalability in EBS. Bidirectional feedback enables participants to assess their current situation and adapt their behavior at runtime if necessary. Our concept encompasses related approaches by treating them as dedicated actions for enforcing requirements for specific properties.
This paper makes the following contributions to support QoI in EBS:

1. a generic model to express malleable requirements and capabilities for arbitrary quality-related properties in EBS at runtime (Sec. 2);
2. algorithms for negotiating requirements at runtime in the middleware by (i) matching expectations and capabilities to identify satisfied, satisfiable and unsatisfiable requirements; (ii) deciding on the requirements to satisfy based on strategies for optimization and load balancing; (iii) enforcing those requirements by adapting participants, the middleware or both (Sec. 3); and
3. runtime support in industry-strength systems illustrated by prototypes implemented in Java within the centralized ActiveMQ JMS messaging broker and the decentralized REDS middleware (Sec. 4).

Related work is discussed in Sec. 5 before Sec. 6 concludes with final remarks.

2 Expectations: support for QoI in EBS

This section describes the challenges supporting QoI in EBS at runtime and our proposed solution using expectations, capabilities and feedback.

2.1 Background: event-based systems in a nutshell

Participants in EBS are independent but cooperative software components with different roles that communicate indirectly using notifications: publishers send

advertisements once before they *publish* notifications to announce the type of event to be provided. Different kinds of data sources can act as publishers: sensors, services or other software components. Subscribers are components that express interest about notifications with a specific type or content by registering *subscriptions* at the middleware. Subscribers and publishers are fully decoupled by the middleware. It matches subscriptions to advertisements and processes notifications from publishers to subscribers based on routing trees, following a many-to-many communication pattern. The middleware can consist of a single, centralized message broker or a distributed network of brokers. Brokers perform efficient en-route filtering and selective forwarding of notifications based on their content. As the message flow is unidirectional, from publishers to subscribers, subscribers are anonymous to publishers, and vice versa.

2.2 Challenges supporting QoI in EBS at runtime

Support for QoI means to deliver information with specific quality-related properties that satisfy individual, sometimes vague, requirements while balancing the costs for provisioning against it [3,18]. EBS are designed for heterogeneous and dynamic populations: publishers and subscribers can join, leave or change at runtime. Multiple publishers can provide information of the same type and content but with different quality-related properties as those depend on each publisher's configuration (e.g., available hardware, setup) and can change dynamically based on a publisher's current context (e.g., enforced energy-saving mode for battery-powered sensors). Information is only propagated by the middleware in a many-to-many fashion, preventing direct negotiation.

2.3 The model of expectations and capabilities

The basic building blocks of our approach are *properties* that characterize information in addition to its content or type. Examples for properties are *precision*, *rate*, *transport latency*, *trustworthiness*, *order*, or *confidence* [20]. Properties do not have to be comparable (e.g., *trustworthiness* vs. *order*) but they can be conflicting due to system constraints (e.g., *rate* vs. *latency* vs. bandwidth). Every property can be modeled over a range or a set of values that apply a total order (ordinal scale) depending on the semantics of the property. For example, *trust* can be modeled over the set $\{\texttt{none}, \texttt{low}, \texttt{medium}, \texttt{high}\}$, with $\texttt{none} < \texttt{low} < \texttt{medium} < \texttt{high}$; *confidence* can be modeled using the range $[0\%; 100\%]$; *transport latency* can be modeled as the number of milliseconds elapsed since publication using the range $[0; \infty]$. Each property can be *improved* by either maximizing or minimizing it, depending on the semantics of the property (e.g., improve latency by minimizing it). A value *dominates* another value of the same property if it improves it (e.g., a confidence of 88% dominates a confidence of 25%, a latency of 300ms dominates a latency of 700ms).

Expectations to express QoI requirements. The context of a subscriber might change at runtime, affecting requirements about quality-related properties of notifications but not those about content or type as expressed in the

subscription (c.f. Sec. 1). We introduce the notion of *expectations* to encapsulate quality-related requirements, enabling subscribers to manage their requirements about quality-related properties at runtime.

Definition 1 (expectation). *An expectation describes a malleable set of requirements that a subscriber has about quality-related properties of information it has subscribed to. Each expectation $\mathcal{X}_i^e$ consists of a set of tuples (p_e, lb, ub) as well as a utility value $\mathcal{X}_i^e.u$ which reflects the individual importance of this expectation for the subscriber and allows a ranking.*

Each tuple in $\mathcal{X}_i^e$ refers to a requirement about a property like *rate*, *confidence* or *latency*: it is defined as a range of values $[p_e.lb;p_e.ub]$ that a subscriber would accept for property p_e and the associated event e. By combining different requirements in a single expectation, each subscriber defines a tradeoff between the ranges of those properties, making the requirements malleable. For example, subscriber M with expectation $\mathcal{X}_1^e = \{(rate, 5, 10), (confidence, 75, 95)\}$ accepts notifications with $\{rate = 7, confidence = 90\}$ as well as notifications with $\{rate = 10, confidence = 80\}$. A subscriber can associate multiple expectations with the same subscription to allow for alternative configurations, ranked by their utility values [27,13]. For example, M needs highly reliable information ($\mathcal{X}_1^e$) but could alternatively do with less reliable information at a higher rate to compensate false-positives/negatives: $\mathcal{X}_2^e = \{(rate,30,45), (confidence,50,60)\}$.

Each expectation has a lifecycle that starts with *registering* it at the broker, making the system aware of the described requirements. Changes in the context of the subscriber can be reflected by changing the lifecycle of a registered expectation by *updating*, *suspending/resuming* or *revoking* it. Registered expectations are active unless they are suspended or revoked. When unsubscribing, all associated expectations are treated as revoked by the broker.

Capabilities to express the system state. In EBS, the system state regarding QoI depends on the extent to which properties are provided by publishers and supported by brokers. We introduce *capabilities* to describe this.

Definition 2 (capability). *A capability describes the extent to which publisher j supports property p_e. Each capability $\mathcal{C}_j^e$ is a tuple $(p_e,lb,ub,cv,cost_{p_e}(x))$ that defines (a) the range of values $[C_j^e.lb; C_j^e.ub]$ publisher j in principle is capable of providing; (b) the value $C_j^e.cv$ within this range that publisher j is currently operating at; and (c) the cost function $cost_{p_e}(x)$ for operating at x.*

A capability describes the current support for a property by a publisher as well as the realizable spectrum of values. Providing p_e at a specific quality comes at a cost [3], captured in $cost_{p_e}()$. A publisher can provide multiple capabilities while the same capability can be provided by multiple publishers with different ranges or costs. Capabilities for some properties like *confidence* are provided only by publishers, others depend on assessment and enforcement by the middleware or a cooperation of publishers and the middleware at runtime (e.g., to support *latency*, *reliability*, *order*). A *capability profile* bundles all capabilities of a publisher for a given event.

Definition 3 (capability profile). *A capability profile $\mathcal{CP}^e_j$ is a set of capabilities $\{\mathcal{C}^e_1, \ldots, \mathcal{C}^e_k\}$ associated with publisher j for events of type e. It consists of capabilities determined by the publisher itself and those determined by the broker.*

A capability profile reflects the full set of capabilities available from a specific publisher for a given event type and can be matched against expectations. Capabilities determinable only by the broker are added at runtime. Capability profiles for the same type of event ($\mathcal{CP}^e$) but associated with different publishers can be heterogenous in terms of the (i) set of properties (e.g., $\mathcal{CP}^e_2 = \{rate \wedge latency\} \subset \mathcal{CP}^e_1 = \{rate \wedge latency \wedge confidence\}$), (ii) ranges, and (iii) current values.

A capability profile's lifecycle starts with *registering* it at the broker and ends with *revoking* it. During runtime, the situation of a publisher might change in a way that requires *updating* registered capability profiles without changing the advertisement. For example, a battery powered sensor runs low on energy and has to switch to an energy-saving mode, decreasing the *rate* of publication; or new resources become available at runtime, improving or adding capabilities (e.g., higher *confidence* due to better contextual information [16]).

Feedback to subscribers and publishers. At runtime, publishers and subscribers are able and willing to adapt their behavior if they get feedback about their actions and the system state. As traditional EBS do not give such feedback at runtime, participants cannot assess if and how they would have to adapt [11]. We introduce *bidirectional* feedback from the middleware to participants to provide them with additional information about their actions and support adaptation at runtime. Subscribers get feedback about the state of their active expectations (*satisfied* or *unsatisfied*). They are informed about the *reason* if an expectation cannot be satisfied by the system at the time. Reasons are expressed as tuples $(\mathcal{X}^e_i, p_e, \alpha)$, describing the value currently provided by the system for each property that is not satisfied. As soon as the expectation can be satisfied, the subscriber is notified about the new state. Publishers receive feedback about each active capability profile's usage together with advice to adapt their publications if necessary. This includes the list of capabilities to adapt together with the required target values, expressed as tuples $(\mathcal{CP}^e_j, \mathcal{C}^e, \beta)$. We consider publishers to be able to adapt automatically at runtime if notified as we show in [13].

3 Negotiating requirements for QoI in EBS

Using expectations to model requirements about QoI and capabilities to describe the corresponding system state, requirements negotiation in EBS can be done automatically at runtime inside the middleware. For every active expectation associated with a subscription, the middleware has to check if it could deliver information with quality-related properties that satisfies the expectation and the associated subscription. This can be possible already with the current state of the system or after adaptation, depending on the capabilities of publishers providing notifications that match the subscription in type or content. In some cases,

however, a requirement cannot be satisfied even after adapting due to limitations of the system or cost constraints and has to be declined. The remainder of this section describes the algorithm for matching expectations to capabilities, outlines how to decide about *satisfiable* expectations and illustrates how suitable reactions are selected at runtime by the middleware.

3.1 Matching expectations to capabilities

As publishers are described by their capability profile in terms of QoI, the whole decision problem is reduced to first a *set-* and then a *range*-matching problem between an expectation $\mathcal{X}_i^e$ and available capability profiles. The result is either a set of publishers with capability profiles already satisfying $\mathcal{X}_i^e$ ($\textsc{Cand}_{\mathcal{X}_i^e}$) or a set of publishers that are capable but would have to adapt ($\overline{\textsc{Cand}}_{\mathcal{X}_i^e}$).

The algorithms for matching an expectation $\mathcal{X}_i^e$ to a set of capability profiles $\{\mathcal{CP}_1^e,\ldots,\mathcal{CP}_l^e\}$ are shown in Fig. 3. The whole process is performed for a single expectation at a time. It can be triggered by a subscriber registering/updating an active expectation or by changes to capability profiles. A changed capability profile requires checking all expectations affected by it.

We define the following terms and relationships for a property p_e of an expectation $\mathcal{X}_i^e$ and a matching capability $\mathcal{C}_j^e$ of a capability profile $\mathcal{CP}_j^e$:

Covered property. A property of an expectation is *covered* if its range overlaps with the range of a matching capability (i.e., $C_j^e.lb \leq p_e.lb \vee C_j^e.ub \geq p_e.ub$) (c.f., Fig. 2 (a)). A property is *fully covered* if its range is enclosed or improved by the range of $\mathcal{C}_j^e$ (i.e., $C_j^e.lb \geq p_e.ub$ for maximization).

Dominated property. A property of an expectation is *dominated* if a matching capability's current value dominates the lower or upper bound of the property. A property that is dominated is also covered whereas a covered property is not necessarily dominated (c.f., Fig. 2 (b)).

Satisfiable expectation. An expectation is *satisfiable* if all its properties are covered by matching capabilities of at least one capability profile.

Satisfied expectation. An expectation is *satisfied* if all its properties are dominated by capabilities of a matching capability profile (c.f., Fig. 2 (c)).

Unsatisfied expectation. An expectation is *unsatisfied* if no matching set of capabilities exists (i.e., $\mathcal{CP}_j^e \subset \mathcal{X}_i^e \vee \mathcal{CP}_j^e \bigcap \mathcal{X}_i^e = \emptyset$) or if at least one property is not dominated by any matching capability (c.f., Fig. 2 (d)).

Deciding *if* an expectation is satisfied, satisfiable or unsatisfied does not require the middleware to compare it with every known capability profile but only with the most promising ones. Thus, each broker B maintains a *SuperSet* $\mathcal{S}_B^e$ per event type e that represents the *skyline* [7] of capabilities available at this broker: For every set of capabilities in $\mathcal{CP}^e$ it contains those capability profiles that are as good or better than all other capability profiles known at this broker in all capabilities and dominating in at least one capability as illustrated in Fig. 4. The *SuperSet* is updated with every change to a capability profile.

An expectation $\mathcal{X}_i^e$ is satisfied ($\mathcal{X}_i^e \in \textsc{Sat}$) if it is dominated by the *SuperSet*, satisfiable ($\mathcal{X}_i^e \in \overline{\textsc{Sat}}$) if covered by it and unsatisfiable ($\mathcal{X}_i^e \in \underline{\textsc{Sat}}$) if not.

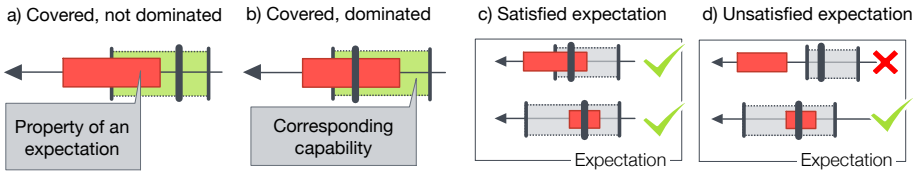

Fig. 2: Relationship between properties and corresponding capabilities.

```
global: SAT, ¬SAT, SAT, CAND, ¬CAND

function MATCH(X_i^e, CP_1^e, ..., CP_l^e)
    State ← unsatisfied
    for all CP_j^e ∈ {CP_1^e, ..., CP_l^e} do
        tS ← CHECKSTATE(X_i^e, CP_j^e)
        switch tS do
            case satisfiable
                ¬CAND_{X_i^e}.add(CP_j^e)
                State ← satisfiable
            case satisfied
                CAND_{X_i^e}.add(CP_j^e)
                State ← satisfied
        end switch
    end for

    switch State do
        case satisfied: SAT.add(X_i^e)
        case satisfiable: ¬SAT.add(X_i^e)
        case unsatisfied: SAT.add(X_i^e)
    end switch
end function

function CHECKSTATE(X_i^e, CP_j^e)
    SAT_{p_e}, ¬SAT_{p_e}, SAT_{p_e} ← ∅
    for all p_e ∈ X_i^e do
        if SATISFIES(C_j^e.cv, p_e) then
            SAT_{p_e}.add(p_e)
        else if COVERS(C_j^e, p_e) then
            ¬SAT_{p_e}.add(p_e)
        else SAT_{p_e}.add(p_e)
    end for

    if SAT_{p_e} ≠ ∅ then return unsatisfied
    else if SAT_{p_e} = X_i^e then return satisfied
    else return satisfiable
end function

function SATISFIES(v, p_e)
    if p_e.minimize then
        if v ≤ p_e.ub then return true
    if p_e.maximize then
        if v ≥ p_e.lb then return true
    return false
end function
```

Fig. 3: Algorithms in pseudocode for matching expectations to capabilities; function COVERS (checking if p_e is covered) is omitted due to space limitation.

In a distributed setup, each broker forwards its *SuperSet* to its directly connected neighbors along the routing tree after modifying it: each contained capability profile is associated with the forwarding broker, masking the identity of the locally known provider (i.e., $\mathcal{CP}_j^e \rightarrow \mathcal{CP}_{b_k}^e$). Broker-related capabilities like *latency* have to be updated as well. Forwarded *SuperSets* are handled like capability profiles registered by local clients at each neighboring broker, starting an iterative update that generates a global skyline at the edge brokers.

Example: Matching in distributed EBS. Consider a distributed EBS with an acyclic routing topology as shown in Fig. 4 (top), consisting of brokers B and C, five publishers and four subscribers for events of type e. Expectations and capabilities are defined over properties p_a and p_b (improvable by minimization). Publishers $P_1 \rightarrow \{\mathcal{CP}_1^e\}, P_2 \rightarrow \{\mathcal{CP}_2^e\}, P_3 \rightarrow \{\mathcal{CP}_3^e\}$ register their capability profiles at broker B (c.f. Fig.. 4 (bottom left)), $P_4 \rightarrow \{\mathcal{CP}_4^e\}$ and $P_5 \rightarrow \{\mathcal{CP}_5^e\}$ at broker C. Broker B forwards its *SuperSet* $\mathcal{S}_B^e = \{\mathcal{CP}_1^e, \mathcal{CP}_2^e\}$ to broker C, masking the identity of P_1 and P_2. Note that $\mathcal{S}_C^e = \mathcal{S}_B^e$ as $\mathcal{S}_B^e$ dominates all other local capability profiles at broker C. At broker C, the sequentially registered

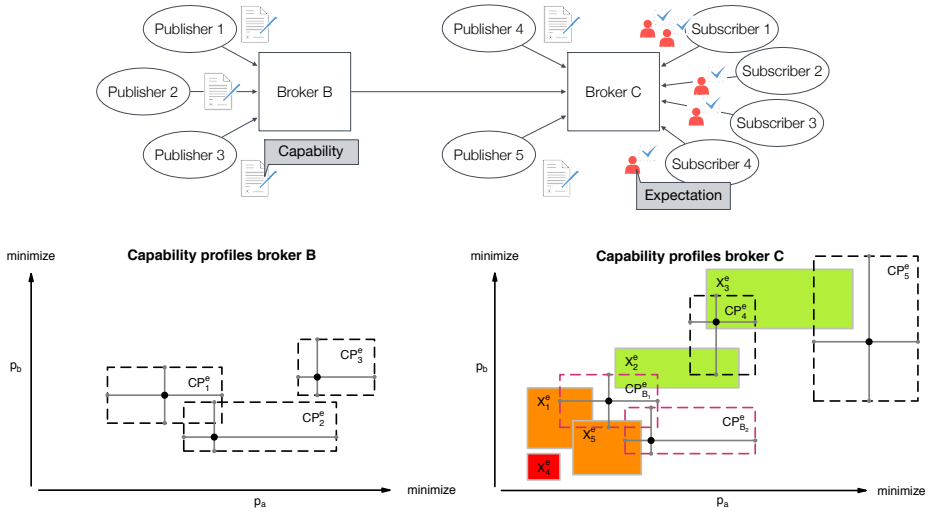

Fig. 4: Example for matching expectations to capabilities in a distributed broker network (top): Broker B forwards the *SuperSet* of its capability profiles (bottom left) to broker C where it is merged with the capability profiles of local publishers ($\mathcal{CP}^e_4, \mathcal{CP}^e_5$) and matched to expectations $\mathcal{X}^e_4$ (not satisfied), $\mathcal{X}^e_2$ & $\mathcal{X}^e_3$ (satisfied), and $\mathcal{X}^e_1$ & $\mathcal{X}^e_5$(satisfiable), (bottom right). Axes show improvement direction.

expectations ($S_1 \rightarrow \{\mathcal{X}^e_3, \mathcal{X}^e_1\}, S_2 \rightarrow \{\mathcal{X}^e_2\}, S_3 \rightarrow \{\mathcal{X}^e_5\}, S_4 \rightarrow \{\mathcal{X}^e_4\}$) are each matched against $\mathcal{S}^e_C$ (c.f., Fig. 4 (bottom right)) using $\textsc{Match}(\mathcal{X}^e_i, \mathcal{S}^e_C)$ (c.f., Fig. 3). This results in: $\textsc{Sat} = \{\mathcal{X}^e_3, \mathcal{X}^e_2\}$ (satisfied), $\overline{\textsc{Sat}} = \{\mathcal{X}^e_1, \mathcal{X}^e_5\}$ (satisfiable) and $\underline{\textsc{Sat}} = \{\mathcal{X}^e_4\}$ (not satisfiable as it is not covered by any capability profile).

3.2 Deciding on satisfiable expectations

The matching algorithm marks an expectation as *satisfiable* if the system could satisfy it by self-adaptation. As this comes at a cost, the middleware has to assess if the expectation should be satisfied or declined. Different optimization strategies can be applied to such a decision problem [21]. For example, we can apply a strategy aiming at pareto-optimal states for subscribers: we decline an adaptation to satisfy $\mathcal{X}^e_i \in \overline{\textsc{Sat}}$ for subscriber i only if another expectation $\mathcal{X'}^e_i$ is already satisfied for subscriber i (i.e., $\mathcal{X'}^e_i \in \textsc{Sat}$) and satisfying $\mathcal{X}^e_i$ would be more expensive than the current state; we decide to adapt in all other cases. Referring to the example in Fig. 4, we assume S_1 to register $\mathcal{X}^e_1$ after $\mathcal{X}^e_3$ has been satisfied. The middleware would approve satisfying $\mathcal{X}^e_1$ by adapting publisher P_1 if $\sum_{p_e}^{\mathcal{X}^e_1} \mathcal{CP}^e_1.cost_{p_e}(p_e.ub) < \sum_{p_e}^{\mathcal{X}^e_3} \mathcal{CP}^e_1.cost_{p_e}(p_e.lb)$

3.3 Select suitable adaptations

The last step of the runtime negotiation process is to adapt the system and give feedback to subscribers. While system adaptation is limited to routing adjustments based on load-balancing strategies for *satisfied* expectations, approved *satisfiable* expectations require further adaptation. In this paper, we focus on runtime adaptation to satisfy an expectation $\mathcal{X}_i^e \in \overline{\text{SAT}}$; adaptation to free up resources or optimize system costs is part of future work.

The system adapts to increase ($\uparrow$) or decrease ($\downarrow$) properties to turn a suitable capability profile $\mathcal{C}_j^e$ into one satisfying $\mathcal{X}_i^e$. This can be achieved by adapting the middleware itself or by using feedback to advise the publisher associated with $\mathcal{C}_j^e$ to adapt. *Actions* define dedicated activities such as `adaptPublisher`. They are associated with properties as tuples (p_e, $\uparrow \vee \downarrow$, `action`, $costs_{action}$). Please note that sequences of actions can be defined as a new action. Alternative actions can be defined by associating multiple tuples for a property. They can have different costs but we assume $costs_{action} = 0$ if there is no alternative action available. For example, *rate* can be decreased by adapting a publisher or by applying a filter at the broker before delivering notifications to the subscriber [13]. This can be modeled in our concept by associating two tuples: (*rate*,$\downarrow$,`adaptPublisher`, $costs_{adaptPublisher}$) and (*rate*,$\downarrow$,`applyFilter`, $costs_{applyFilter}$).

We are currently selecting the least expensive action for a property to apply. Using other selection strategies at runtime is out of scope of this paper.

4 Implementation

Runtime support for QoI in EBS using expectations and capabilities is realized by extending the middleware with an `ExpectationController` and providing additional handlers to participants as shown in Fig. 5. We have implemented two prototypes in Java, extending the ActiveMQ JMS messaging broker[3] and the distributed REDS middleware[4]. We chose these two platforms for their different features: ActiveMQ is representative of an industrial-strength messaging system focussing on high performance, while the modular REDS systems allows us to exploit routing strategies and broker topologies for adaption. Both systems are easy to extend without affecting existing code. We use our prototypes to support QoI at runtime within the open-source monitoring system Ganglia[5] [13]. In this paper, we focus on describing the key components for a single broker setup.

4.1 Broker extension: ExpectationController

We require access to the broker state for monitoring the system and to apply broker-related reactions like filtering messages or routing adaptation [12,13].

[3] `https://activemq.apache.org`
[4] `http://zeus.ws.dei.polimi.it/reds/`
[5] `http://ganglia.sourceforge.net/`

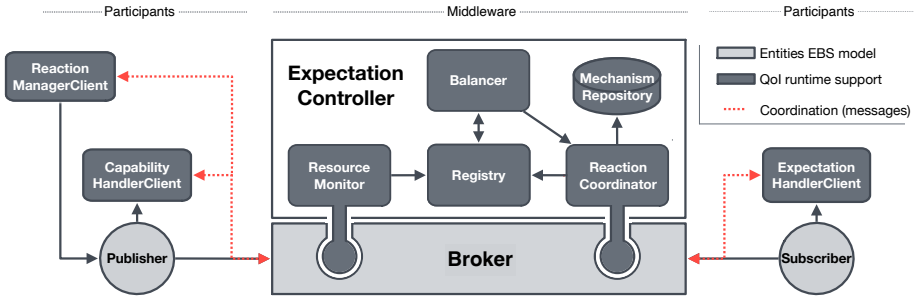

Fig. 5: Runtime support for QoI in EBS with expectations and capabilities showing additional components (dark gray) for participants and middleware (gray).

Thus, we provide `ExpectationController` as a plugin using `BrokerPluginSupport` on ActiveMQ and as an extended `NodeDescriptor` class defining a new broker type on REDS. Other components are implemented in an platform-agnostic way while platform-specific messages are used to communicate with participants. An `ExpectationController` consists of five key components (c.f. Fig. 5 (centre)):

RessourceMonitor monitors the broker's state and the system's population, reporting changes to the Registry.

Registry stores all expectations and capabilities registered at this broker with the definitions of available properties and their matching. Changes trigger a negotiation of requirements at the Balancer.

Balancer matches expectations to capabilities (c.f., Sec. 3) while applying different optimization strategies. Triggers ReactionCoordinator upon completion.

ReactionCoordinator selects applicable actions from the MechanismRepository and coordinates their execution by adapting the broker, advising selected publishers to adapt using feedback or notifying subscribers.

MechanismRepository stores available actions for specific properties (c.f., Sec. 3.3). Actions are objects implementing generic or platform-specific activities.

4.2 Handlers for participants

We provide participants with handlers to deal with feedback by the middleware and use platform-agnostic APIs for managing the lifecycle of expectations and capabilities: ExpectationHandlerClient allows subscribers to store, load, register, revoke, update, suspend or resume expectations. CapabilityHandlerClient enables publishers to store, load, register, revoke or update capabilities and access their usage statistics; publishers can register to be triggered by adaptation advices. Otherwise, an optional ReactionManagerClient adapts its associated publisher if advised by the ReactionCoordinator. For example, within our Ganglia scenario we implemented it as a wrapper that changes the configuration of each `gmond` publisher on the fly before restarting it, realizing adaptation within 26ms.

Expectations and capabilities are stored in XML while property definitions are separately stored using a key-value syntax. We chose these open formats for maximum portability. We provide a parser to process instances of expectations and capabilities with their property definitions in Java as well as a graphical editor to support the user.

5 Related work

Work done by Keeton et al. [18] on general considerations about information quality and by Wilkes [27] on balancing requirements with consumers' utility has highly influenced our work; Behnel et al. [4] and Appel et al. [1] identify a basic set of quality guarantees and the levels of abstractions specific to EBS.

Our model has been inspired by complementary work on specifying and categorizing QoI for sensor networks: Perera et al. [20] support users in searching for sensor data sources using ontologies while the CommonSens middleware for assisted living by Soberg et al. [26] automatically selects sensors based on their domain-specific capabilities. Hossain et al. [16] and Bahjat et al. [3] propose frameworks to quantify QoI in IoT applications focussing on properties like uncertainty, precision, integrity or timeliness of detection. Bisdikian et al. [6] try to separate inherent quality attributes from application-specific ones as do Sachidananda et al. [23]. Our concept generalizes these application- and domain-specific properties, allowing requirements and capabilities to be expressed in a consistent and information-centric way.

We see most approaches proposed in the domain of EBS as complementary to our concept as they provide mechanisms to enforce dedicated quality-related properties that we can model using expectations: several systems address the issue of quality of service (QoS), focussing on network-specific properties like latency or jitter. We refer to [5] for an extensive overview and a more detailed discussion. Directly related to our work are the reactive middleware systems IndiQoS, as proposed by Carvalho et al. [9], Adamant by Hoffert [15] and Harmony by Yang et al. [28]. They support requirements about latency, reliability and bandwidth but focus on a closed set of requirements that is resolved on the transport protocol level only and omit enforcing publisher-related properties. We expand the scope of runtime support to include the enforcement of publisher-related properties by runtime adaptation based on feedback and allow subscribers to expose individual tradeoffs between requirements.

Related topics actively researched on in the area of (cloud-based) SOA are concepts for service selection as well as the negotiation of quality requirements and service-level agreements at runtime. We refer to [19,25] for a detailed discussion due to space limitations and would like to focus on two related contributions: Kattepur et al. [17] define a QoS metric similar to properties in expectations. However, they focus on interactions in heterogeneous SOA choreographies while expectations are information-centric; Pernici et al. [21] use fuzzy parameters for deciding on web service adaptation. Those approaches, however, are based on direct contracts between service providers and service consumers, often assuming

the existence of explicitly modeled workflows or a central authority for coordination. They are not directly applicable to the indirect communication model of anonymous EBS. Integrating them with our concept is part of ongoing work.

6 Conclusion, ongoing and future work

Event-based systems (EBS) complement SOA and enable enterprises to react to meaningful events in a timely manner. While quality of information (QoI) is crucial in these information-centric systems, it is supported only to a limited degree in today's EBS. We introduce the concept of **expectations** as a generic model to express, negotiate and enforce requirements about QoI in EBS at runtime. Instead of providing a fixed set of supported properties, our solution enables participants to define and manage requirements about arbitrary quality-related properties while exposing individual tradeoffs. Requirements are negotiated and enforced at the middleware by adapting data sources and brokers based on different optimization strategies and platform-specific mechanisms. Ongoing work focusses on evaluating our prototypic implementations in terms of performance and scalability using SPEC Research FINCoS[6] and the jms2009-PS benchmark [24]. Future work investigates interdependent and conflicting properties (e.g., adapting the system to support *order* for satisfying one expectation might lead to increased *latency*, violating other expectations). We also plan to extend our model to handle *composite properties* of expectations and capabilities such as *alternatives* [4] (i.e., notifications have to be provided by a number of different publishers all supporting a specific set of properties). Security and privacy aspects are important but orthogonal to our approach and currently out of scope.

Acknowledgements We thank Stefan Appel, John Wilkes and Kimberly Keeton for their feedback on our work; Pascal Kleber and Erman Turan for their support in building the prototypes. Funding by German Federal Ministry of Education and Research (BMBF) under research grants 01|C12S01V and 01|S12054.

References

1. Appel, S., Sachs, K., Buchmann, A.: Quality of service in event-based systems. In: 22nd GI-Workshop on Foundations of Databases (GvD) (2010)
2. Araujo, F., Rodrigues, L.: On QoS-aware publish-subscribe. In: ICDCSW (2002)
3. Bahjat, A., Jiang, Y., Cook, T., La Porta, T.: Quality of information functions for networked applications. In: PERCOM Workshops (2012)
4. Behnel, S., Fiege, L., Mühl, G.: On quality-of-service and publish-subscribe. In: ICDCS Distributed Computing Systems Workshops (2006)
5. Bellavista, P., Corradi, A., Reale, A.: Quality of service in wide scale publish/subscribe systems. IEEE Communications Surveys & Tutorials (99), 1–26 (2014)
6. Bisdikian, C., Kaplan, L., Srivastava, M.: On the quality and value of information in sensor networks. ACM Transactions on Sensor Networks 9(4), 39:26 (2010)

[6] http://research.spec.org/tools/overview/fincos.html

7. Borzsony, S., Kossmann, D., Stocker, K.: The skyline operator. In: ICDE (2001)
8. Buchmann, A., Appel, S., Freudenreich, T., Frischbier, S., Guerrero, P.: From calls to events: Architecting future BPM systems. In: BPM (2012)
9. Carvalho, N., Araujo, F., Rodrigues, L.: Scalable QoS-based event routing in publish-subscribe systems. In: Network Computing and Applications (2005)
10. Frischbier, S., Gesmann, M., Mayer, D., Roth, A., Webel, C.: Emergence as competitive advantage - engineering tomorrow's enterprise software systems. In: ICEIS (2012)
11. Frischbier, S., Margara, A., Freudenreich, T., Eugster, P., Eyers, D., Pietzuch, P.: ASIA: application-specific integrated aggregation for publish/subscribe middleware. In: Middleware 2012 Posters and Demos Track (2012)
12. Frischbier, S., Margara, A., Freudenreich, T., Eugster, P., Eyers, D., Pietzuch, P.: Aggregation for implicit invocations. In: AOSD (2013)
13. Frischbier, S., Margara, A., Freundenreich, T., Eugster, P., Eyers, D., Pietzuch, P.: McCAT: Multi-cloud Cost-aware Transport. In: EuroSys Poster Track (2014)
14. Hinze, A., Sachs, K., Buchmann, A.: Event-based applications and enabling technologies. In: DEBS (2009)
15. Hoffert, J., Schmidt, D.: Maintaining QoS for publish/subscribe middleware in dynamic environments. In: DEBS (2009)
16. Hossain, M.A., Atrey, P.K., Saddik, A.E.: Context-aware QoI computation in multi-sensor systems. In: MASS (2008)
17. Kattepur, A., Georgantas, N., Issarny, V.: QoS analysis in heterogeneous choreography interactions. In: ICSOC (2013)
18. Keeton, K., Mehra, P., Wilkes, J.: Do you know your IQ? A research agenda for information quality in systems. ACM SIGMETRICS Performance Evaluation Review 37(3), 26–31 (2010)
19. Kritikos, K., Pernici, B., Plebani, P., Cappiello, C., Comuzzi, M., Benrernou, S., Brandic, I., Kertész, A., Parkin, M., Carro, M.: A survey on service quality description. ACM Computing Surveys 46(1), 1 (2013)
20. Perera, C., Zaslavsky, A., Christen, P., Compton, M., Georgakopoulos, D.: Context-aware sensor search, selection and ranking model for internet of things middleware. In: Mobile Data Management (2013)
21. Pernici, B., Siadat, S.: Adaptation of web services based on QoS satisfaction. In: ICSOC (2010)
22. Pietzuch, P., Eyers, D., Kounev, S., Shand, B.: Towards a common API for publish/subscribe. In: DEBS (2007)
23. Sachidananda, V., Khelil, A., Suri, N.: Quality of information in wireless sensor networks: a survey survey. ICIQ (2010)
24. Sachs, K., Appel, S., Kounev, S., Buchmann, A.: Benchmarking publish/subscribe-based messaging systems. In: Database Systems for Advanced Applications: DASFAA 2010 International Workshops: BenchmarX'10 (2010)
25. Shi, Y., Chen, X.: A survey on QoS-aware web service composition. In: Multimedia Information Networking and Security (2011)
26. Soberg, J., Goebel, V., Plagemann, T.: CommonSens: personalisation of complex event processing in automated homecare. In: ISSNIP (2010)
27. Wilkes, J.: Utility functions, prices, and negotiation. HP Labs HPL-2008-81 (2008)
28. Yang, H., Kim, M., Karenos, K., Ye, F., Lei, H.: Message-oriented middleware with QoS awareness. In: ICSOC (2009)