Tutorial: Policy Enforcement within Emerging Distributed, Event-Based Systems

Jatinder Singh
Computer Laboratory
University of Cambridge, UK
jatinder.singh@cl.cam.ac.uk

Jean Bacon
Computer Laboratory
University of Cambridge, UK
jmb25@cl.cam.ac.uk

David Eyers
Dept. of Computer Science
University of Otago, NZ
dme@cs.otago.ac.nz

ABSTRACT

Computing is becoming increasingly ubiquitous. To fully realise the potential of emerging distributed systems, it must be possible to *manage* and bring together (*coordinate*) system components in various ways—perhaps for purposes and in circumstances not contemplated by their designers. Therefore, we believe that the application logic embodied in components should be separated from the policy that specifies where, how and for what purpose they should be used.

This paper explores how supporting infrastructure can enable policy, representing high-level (user) or systems concerns, to drive system functionality. SBUS is a middleware that supports secure, dynamic reconfiguration, providing the means for policy enforcement across system components. We present SBUS to demonstrate the practical aspects and design considerations in a) making infrastructure policy-compliant, and b) leveraging the dynamic policy enforcement capabilities to achieve particular functional goals.

Categories and Subject Descriptors

H.3.4 [**Systems and Software**]: Distributed Systems

Keywords

middleware, policy, enforcement, reconfiguration, service, event-based systems, pervasive, security, internet of things

1 Introduction

Trends in computing are towards an ever increasing number of system *components*,[1] that bring new functionality, and produce and consume more data than ever before. The key challenge is management: the benefits come from the increased ability to analyse, process and react to data when and where appropriate. It follows that components may no longer operate in the context of a single application, but may be relevant to, and across, a range of services.

To fully realise the potential of emerging distributed systems, it must be possible to *manage* and bring together (*coordinate*) components in various ways—perhaps for purposes and in circumstances not contemplated by their designers. For example, components developed for monitoring patients in hospital contexts are increasingly being used in personal health and wellbeing monitoring. This requires the ability for cross-component control.

Therefore, we believe that the application logic embodied in components should be separated from the *policy* that specifies where, how and for what purpose they should be used. In addition to flexibility, abstracting policy concerns in this way also lowers the barrier for entry (leaving component designers to focus on core functionality, rather than on policy concerning potential uses and interactions), while fewer enforcement points can facilitate policy management and reduce the chance for errors.

Middleware today tends to manage communication and resources pertaining to an application, at the application's request. In emerging systems, the environment is highly dynamic and variable; it is not just the infrastructure and resources underlying applications that must be configured and managed, but also the applications themselves.

Moving forward, the major requirement for application-supporting infrastructure is the ability to reconfigure components dynamically in response to changing circumstances; for example, due to some failure, or on detecting a health emergency or a change in location or underlying network.

SBUS [27] is a middleware that has evolved to provide such a capability, in a secure, open decentralised manner. This allows policy enforcement components to manage and control components (and thus various systems), when and where appropriate—in accordance with user-defined policy.

In this paper, we present the SBUS middleware, and use it to explore concerns regarding a) the mechanisms for making infrastructure policy-aware and compliant, and b) leveraging the dynamic policy enforcement capabilities to achieve particular functional goals. We describe the role of policy engines in reconfiguring systems, present two such policy engine implementations aimed at different concerns, and provide practical examples to illustrate the design considerations of policy-based systems. In addition, we also consider more general requirements on emerging distributed systems, including the need for security—particularly where policy-based reconfiguration is involved, and the need for middleware to support a range of interaction paradigms, including stream-based as well as the more traditional request-reply.

[1] We define *component* broadly to refer not only to a system-level service, as is common in this area, but also to include whole applications and services, as well as parts thereof.

DEBS'14, May 26–29, 2014, MUMBAI, India.
ACM 978-1-4503-2737-4/14/05.
http://dx.doi.org/10.1145/2611286.2611310.

2 The SBUS middleware

SBUS [27] is a middleware for managing interactions in dynamic, pervasive environments. SBUS was originally developed for city-wide transport monitoring [14, 2], and has been extended for healthcare and assisted living scenarios [26, 4].

SBUS is an open systems framework for securely reconfigurable components. It has standard middleware attributes, such as a type system, a transfer syntax, a security regime, and resource discovery components. Other features are its support for multiple interaction patterns and for endpoint type negotiation.

Crucially, SBUS supports dynamic reconfiguration for the runtime management of interactions/information flows. Not only can applications reconfigure themselves, but they can *control the interactions of others* (subject to authorisation checks). This functionality is critical, because it provides *the mechanism for policy to effect coordination and control*.

This section presents SBUS and its reconfiguration capabilities, which provides the basis for policy-based management in emerging distributed systems. We later explore how these are leveraged to enable a policy-enforcing capability.

2.1 General overview

SBUS is a messaging middleware that is designed to be open and flexible. No structure is imposed on system design, but rather, it provides the 'building blocks' to facilitate a variety of structures to suit those using the middleware.

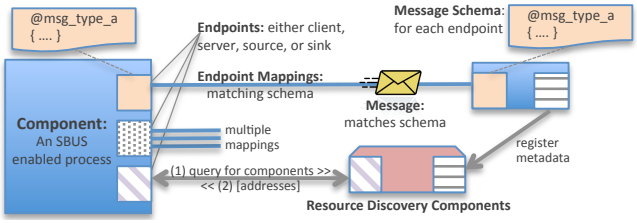


Figure 1: SBUS Conceptual Overview

SBUS is a component-based messaging middleware. In SBUS, a *component* represents a process (i.e. application, service or part thereof—in line with our previous definition) whose communication is managed by SBUS. Communication occurs through *endpoints*, which are essentially typed communication ports. A component will have a number of endpoints, which can be connected *(mapped)* to endpoints of other components to enable communication (see Fig. 1).

SBUS supports a range of basic interaction paradigms that support both request-response and stream-based communication (Fig. 2), which is important as emerging systems will have requirements for both forms of communication. An endpoint is defined to take an *interaction mode*: *client* (request) or *server* (response); or for streams, either a *source* (producer) or *sink* (consumer). Mappings occur between endpoints whose interaction modes correspond: sources with sinks, clients with servers.

Although SBUS is inherently decentralised, it also provides the building blocks to enable other interaction models. Therefore, while communication is peer-to-peer, pub/sub (event-bus) and message-queue brokers are easily implemented, thus enabling indirect and asynchronous communication. The SBUS distribution includes a pub/sub broker, its core functionality implemented in only a few lines of code.

SBUS is a messaging middleware, where communication

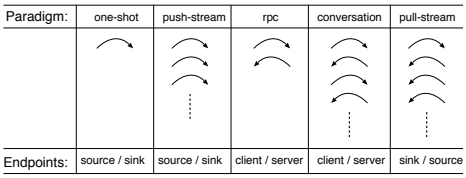


Figure 2: SBUS interaction paradigms

is data centric. A *message* encapsulates data of a particular type, and is often used to represent the details of an event. Message types are described in *LITMUS* [14] (Fig. 3), allowing expressive definitions including arrays and lists. An endpoint is associated with a schema describing the message type(s)[2] it handles. Mappings may only occur between type-compatible endpoints.

`int name`	`dbl name`	Integer, Floating point,
`flg name`	`txt name`	Flag (boolean), Text string,
`clk name`	`loc name`	Date and time, Location,
`bin name`		Binary data,
`[ elt ]`	`<elt1...eltN>`	Optional element, Choice,
`-`	`* Foo bar`	Unnamed elt, Comment,
`@elt`	`^label name`	Type defn, Type reference,
`@"filename"`		Import types from file,
`name { elt1 ... eltN }`		Structure,
`name ( elt )`		List of elt,
`name (+ elt )`		Non-empty list,
`name (N elt )`		Array of N elements,
`name < #val1 ... #valN >`		Enumeration,
`name1 + ... + nameN`		Multiple declaration

Figure 3: LITMUS type-representation syntax

Once a mapping is established, messages may be transmitted. SBUS validates each message against the relevant schema. To reduce network overhead, messages between components are binary encoded. LITMUS type hashes are also included, making messages self-identifying. This allows a fast (probabilistic) type check that is stronger than a type-ID, and removes the need for a central type authority. Applications receive message data directly through library functions that can access typed attributes, and/or through an XML representation.

SBUS allows the definition of content-based filters to select (limit) the messages transmitted. Filters are connection specific: different filters can apply to different mappings on the same endpoint. They are evaluated in the context of a message, and can be changed at runtime. The language is highly expressive [14]. Filters aid efficiency by avoiding unnecessary transmissions, and also security, by preventing certain consumers from receiving particular messages.

2.2 Resource discovery

Connections can only be established where network addresses are known. To assist this, a component can register its *metadata*—describing itself, its function, and data handled—with a *Resource Discovery Component* (RDC). RDCs maintain a directory of active (registered) components, to enable the runtime discovery of components of interest.

The RDC provides a component lookup service that returns the addresses of components whose metadata match the criteria specified in a *map-constraints* query. Such queries are made up of any number of constraints, which tend towards two categories:

Identity: Concerns general aspects of the component, such as its class (named-type), instance-name, owner, author (developer), or public key (identifying one specific component).

[2]Client/server interactions have one message type associated with the request, and another for the response.

Original Message

```
<place>
   <coordinates>
      <position>
         <longitude>0.091732</longitude>
         <latitude>52.210891</latitude>
         <height>19</height>
      </position>
   </coordinates>
   <placenm>somewhere</placenm>
</place>
```

Repackaged Message

```
<location>
   <gps>
      <longitude>0.091732</longitude>
      <latitude>52.210891</latitude>
      <altitude>19</altitude>
   </gps>
   <city>""</city>
</location>
```

Figure 4: Example schema negotiation, where the `<position>` type is found and converted to `<gps>` for a particular component.

Data: Concerns the data (endpoint schemata) that the component offers. Generally, mappings will only occur between matching endpoints.

Some data constraints enable *schema negotiation*, which attempts to balance the benefits of strong typing with the flexibility required by emerging systems, e.g. in supporting mobility (see §3.1.1). Two operators, `has` and `similar`, exist to negotiate the local endpoint's schema with another component. The operators take an attribute of the local LITMUS schema as an argument, typically a structure, to find a suitable (comparable) endpoint on the peer. `Has` ensures that both attribute names and types match; `similar` compares only the attribute types.[3] If agreement is possible, SBUS will enable a connection, and automatically repack incoming messages into the local format (Fig. 4). This functionality is useful where policy designers have some indication about the components that will need to interact, but whose data models do not directly correspond.

Of course, the location of an RDC is also important. In addition to prior knowledge and/or running at a well-known address, each component maintains a list of RDCs with which it interacts. This can be changed at runtime, meaning components can be told of the relevant RDC addresses.

In practice, there will be a number of RDCs operating within specific scopes. Some may be federated, perhaps replicating the directory information across a global enterprise. Others might operate in a far more limited scope, e.g. dealing only with components in a ward or patient's house. Any structuring will be defined by the particular application domain. For example, one could imagine a number of RDCs in the same physical space: several cooperating to manage large-scale distributed application, and one managing end-user services.

RDCs are optional components, existing only to assist the management of components. Discovery is also possible through *inspection*, where components send probe messages to connected peers to obtain a view of available components by trawling a connectivity graph.

2.3 Disconnections and Failures

In a dynamic environment, connectivity, disconnections and failures must be managed. SBUS does not attempt to prescribe how disconnections are handled. This is because the proper response will depend on the application, environment and circumstances. Thus, SBUS provides the mechanisms to enable policy to detect and respond to disconnections/ failures as appropriate.

SBUS, through RDCs, can automatically connect a component to a peer serving similar data [14]; although this is not always appropriate. Again, SBUS provides the building blocks for more complex scenarios, such as those requiring information outside component state (e.g. "two As must be connected to a B"). This allows external components, such as policy engines, to manage the mappings. Such an approach facilitates mobility management (§3.1.1), e.g. where a person moves between active environments. To assist with failures each endpoint appends a sequence number to each message and mapping, enabling message buffering and replay where necessary. Again, broker components (such as message-queues) can be built to manage data distribution and consumption at a higher level.

It is important that RDC registries are accurate. SBUS manages this in two ways: 1) components automatically inform RDCs when a peer unexpectedly disconnects; 2) each RDC uses a (configurable) periodic 'heartbeat' to ensure the liveness of the registered components.

2.4 Security

The SBUS security model enables governance and control over middleware operations. These work to complement any application-level security mechanisms that may be in place, such as login services, biometric readers, etc.

2.4.1 Access control

Middleware must provide the means to **protect data in transit**. Control is intuitive in SBUS, given communication is peer-to-peer; as opposed to an event-bus or other shared communication channel where potentially a number of components see the same message. SBUS uses *Transport Layer Security* [11] to establish a secure communication channel between components. Components must exchange and validate certificates before any communication to protect both messages and protocol state. Unsecured (certificateless) communication is possible, but both peers are made aware and must agree to interact in such a manner.

Regarding **access control**, each endpoint is associated with an *access control list (ACL)* that describes the components that may interact. On connection, each component will consult its ACL to decide if the peer is authorised, a mapping established only if each component authorises the other. Should privileges change, any active mappings are re-examined, with the connection closed if the particular peer is no longer authorised.

Access control policy refers to the metadata of the component, currently its `class`, `instance name` and/or `public key`.[4] This allows the ACL to vary in specificity, from applying to a specific component or a larger set. For strong authentication, and to ensure that access control targets the correct components, we couple identity information to certificates, verifiable through TLS. This results in a regulated namespace, which can be useful in particular application domains, e.g. in healthcare to regulate the components whose identities are tied to a particular patient-ID [26]. If a component cannot be authenticated (i.e. it lacks a certificate), it may only interact with open (world-readable) endpoints.

To control access to particular data, filters can be imposed on a mapping to select the messages suitable for transmission. This works as an authorisation rule that is evaluated on message content.

[3] See [12] for a survey on metadata interoperability.

[4] This can easily be extended to include other attributes of component metadata, or perhaps tokens relating to other authentication systems.

2.4.2 Secure discovery

Sometimes there are situations where knowledge component's existence may be sensitive, e.g. in a healthcare context, those relating to sexual health. Components can hinder discovery by electing not to register with an RDC or limiting inspection operations, though this may preclude important interactions.

RDCs have two mechanisms for controlling discovery operations. Firstly, an RDC has access control policy defining who may register and query. Secondly, an RDC mirrors the ACLs of its registered components, which is enforced against the results of a discovery query. This means a component that issues a discovery query will only receive results for the components whose endpoints it is authorised to access.

To regulate inspection operations, a component a) maintains access control policy restricting those that can issue inspection queries, and b) can dictate whether its peers are allowed to reveal its existence in inspection operations.

Though these approaches provide *security by obscurity*, protecting discovery mechanisms helps prevent accidental disclosure, and adds an extra hurdle for the malicious. In any case, the access control mechanisms still operate to protect the data, even if the component is known.

2.5 Database integration

Most systems require persistence, with relational databases being commonplace. Interacting with a database requires much knowledge about its specifics, including the vendor, its location, relevant driver(s), and *the structure of the data contained*. This is inappropriate for the environment described, where often a component will not know in advance if, when, and where data is persisted, nor how data is represented in the database, which underpins the ability to query.

We therefore developed *SBUS-PG* to allow components to take advantage of database functionality, purely through the use of messages. SBUS-PG integrates SBUS and PostgreSQL, by associating a database instance with a component *(proxy)* to manage SBUS interactions. It can automatically translate the relevant database objects into SBUS message types, accounting for relations, e.g. where foreign key constraints translate to nested LITMUS structures, custom types, etc. (Fig. 5). Marshalling between tuples and messages is facilitated as the systems are written in `C/C++`.

PostgreSQL Table

```
CREATE TYPE
   complex
AS (Re float8, Im float8);

CREATE TABLE
FourierTransform (
   'label' varchar(20) NOT NULL,
   'coefficients' complex[] NOT NULL,
   'ts' timestamp );
```

SBUS Type Definition

```
@FourierTransform {
   label txt
   coefficients (
      coefficients {
         Re dbl
         Im dbl }
   )
   [ ts clk ]
}
```

Figure 5: Mapping a table, with a complex type, to a LITMUS schema

This integration exposes core database functionality to other SBUS components. `Insert`s involve having relations correspond to a sink endpoint, into which any messages received on that endpoint are inserted. `Select`s are implemented to provide continuous (source/sink) or single query (client/server) functionality. For the continuous query functionality, an SBUS source endpoint is created and linked to a relation, so that subsequent inserts are sent as messages to the mapped components. This is similar to pub/sub. For client/server, stored procedures are used to implement a traditional, single SQL query. A stored procedure is defined to take query variables as parameters and return the query result as a set of tuples. The procedure is coupled with an SBUS server endpoint, so the request corresponds to the query variables, and the response is the messages representing the tuples returned from the procedure. `Updates` and `deletes` are possible, but must be implemented through custom stored procedures that can properly account for the semantics of such operations.

SBUS-PG allows components to interact with the database without the need to know the relational data structures, nor database specifics. The data available to/from the database is immediately visible through the types exposed by the integration component. While not part of SBUS itself, SBUS-PG facilitates dynamic component interactions with layers of persistence, through a seamless interface (cf. database drivers). This will be important for emerging systems. §3.2 describes how a policy engine was built from SBUS-PG.

2.6 Runtime reconfiguration

A key feature of SBUS is its capacity for runtime configuration, the API for which is presented in Table 1. Components use this to change their state. As part of a reconfiguration operation, all related activities are performed; for example, changing a privilege on an endpoint will close the connections no longer authorised, and the RDC will be told to update the relevant ACL it mirrors.

`map(map_params)`	Establishes a mapping between endpoints.
`unmap(map_params)`	Terminates a mapping.
`divert(divert_params)`	Moves an endpoint's mapping(s) to another component.
`subscribe(filter)`	Changes a mapping's content-based filter(s).
`privilege(ac_policy)`	Alters an endpoint(s)' access policy.
`rdc_addr(addresses)`	Changes the RDCs that a component uses.

Table 1: SBUS reconfiguration functions

Third-party initiated reconfiguration

The key feature of SBUS in enabling policy-enforcement is the ability for third-party, or remote reconfiguration. SBUS essentially allows a component's reconfiguration API to be remotely invoked; which means one component can instruct another on how to behave.

This is implemented by each component having, as default, a set of *control endpoints* that correspond to the reconfiguration API (Table 1). When a component receives a message on a control endpoint, it performs the related reconfiguration operation (message content defining the parameters), in the same way as if the operation was self-invoked.

The process is illustrated in Fig. 6, where a component instructs another to undertake a mapping (1). In this example, as the map instruction contains map constraints (a component metadata query), it implicitly forces a RDC query to find the address of the component to map to (2), after which the mapping is established (3).

Given the power of third-party reconfiguration, the access control regime also applies to control endpoints (just as any other), to ensure reconfiguration instructions are only actioned when received from a trusted peer.

Third-party initiated reconfiguration is crucial for enabling

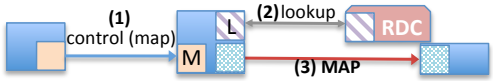


Figure 6: Third-party initiated mapping

the type of policy enforcement described, as it provides the mechanism to directly control components and their interactions from outside their application logic.

2.7 SBUS summary

SBUS is a component-based messaging middleware that aims to be sufficiently flat and flexible to support a range of concerns and environments. It enables secure, type-safe, client/server and stream-based interactions, facilitates discovery, and can seamlessly integrate with databases. Importantly, it is dynamically reconfigurable, where operations can be instigated by components external to the action. This paves the way for policy to control and reconfigure components, when and where appropriate—functionality that is fundamental to supporting emerging distributed event-based systems.

3 Policy enforcement

Policy encapsulates a set of concerns, defining the actions to take in particular circumstances to effect some outcome. With respect to middleware, policy has traditionally concerned issues of network management, resource allocation and/or quality of service. However, as discussed, in emerging distributed systems policy may also relate to high-level (or user) concerns. This involves both managing and *coordinating* components, dictating how they behave, and when they interact.

Achieving this requires control mechanisms that can affect a component from *outside its application logic*. Again, such an approach enables new functional possibilities as components can be used/reused in various ways, determined by users rather than only the original developers. Further, the abstraction of policy specifics lifts the burden on components of maintaining their own internal policy representation, and removes the unrealistic and unmanageable requirement for component designers to account for all potential uses and operating environments for their component, as well as all possible components with which it may interact. This works to facilitate policy management, while fewer points of enforcement can reduce the propensity for errors.

As middleware operates across applications, it is highly amenable to implementing such a policy enforcement capability. Middleware supporting third-party reconfiguration, like that provided by SBUS, enables flexible policy enforcement since it allows a component to be instructed by potentially any other (subject to privilege) to perform a particular middleware action.

In SBUS, the reconfiguration capabilities (Table 1) can generally be used to: a) directly effect an interaction, e.g. force a connection or disconnection; or b) establish the groundwork to allow a possible future interaction, e.g. changing privileges or visibility from an RDC.

The SBUS discovery mechanisms also help flexible policy definition by allowing policy to refer to components: a) explicitly, i.e. a particular component, perhaps at a particular address; or b) more generally in terms of desired properties, such as any component(s) in the environment dealing with particular data. This means the components for which policy applies can be determined at runtime, enabling policy that can, for e.g., find and connect any data source of type `X` to the local datastore. As this paper focuses on middleware concerns and enforcement, we do not discuss policy authoring considerations here, though we have previously [25, 24].

3.1 Policy engines (PEs)

Often, policies are represented in terms of *event-condition-action* rules [9], where particular events, in certain situations, trigger an action, e.g. informing medics in a medical emergency. In SBUS, such an event can be encapsulated and communicated within a message. The event/trigger might result from a single message indicating some occurrence, some processing (complex-event detection), or some other happening (where context may occur at a higher level).

Policy actions represent the response to take. With respect to a policy-based middleware, actions tend towards three categories:

Reconfiguration: Executing a reconfiguration operation on particular components (i.e. Table 1).

Message production: Generating messages (representing events) to communicate some information.

Policy management: Policy applies in a particular context. Therefore, state changes can alter the set of applicable (or active) policies, e.g. relaxing privacy policies in a medical emergency to facilitate the most appropriate response.

In SBUS, any component can execute these sorts of actions, because any component can produce a message, and thus, has the ability to reconfigure another.

A *policy engine* (PE) is a service that encapsulates, and enforces, a set of policies. In practice, we expect PEs to maintain sets of related policies, e.g. to a particular user, organisation, physical space, service contract, and so forth.

We now present two policy engines that we have implemented, each with a different focus, to show how policy can be practically enforced.

3.1.1 Mobile Policy Engine (MPE)

The *Mobile Policy Engine* (MPE) is a component built for Android mobile devices. Its aim is to support mobility, by managing the interactions between the components on the device and in its (physical) environment.

An MPE maintains a set of policies (rules) that execute on particular events. Events correspond to endpoints, such that receiving a message triggers the policy actions defined for that event. SBUS content-based filters can further refine the circumstances in which the rules apply, enabling fine-grained policy. The MPE was integrated with the *Android Remote Sensing* Service,[5] allowing rules to be defined for any of the 40+ sensor/event streams from an Android device.[6]

Our motivation for developing the MPE was to manage mobility: components will need to adapt as the device moves between different operating environments. This aims at providing more than just seamlessness, as it is often necessary to change functionality and the associated goals given the new environment, e.g. certain things should happen when someone enters their home, which are different from when

[5] `https://play.google.com/store/apps/details?id=com.airs`

[6] Other events can be integrated simply by adding a new sink endpoint to the MPE, against which policy can be defined. This allows the integration of complex event detectors, signal processing modules, advanced positioning systems, etc.

they enter a shopping mall. Our focus was therefore on related events, such as a change in network and/or the presence of new RDCs. Policies would automatically connect device-components to relevant components in the new environment, and inform applications of the change to allow them to adapt.

This work was part of the motivation for schema negotiation (described in §2.2), where components want particular data, but do not necessarily know the identity specifics or availability of the components in the new environments.

3.2 Database policy engine (DBPE)

Databases are ideal for integrating PE functionality. This is because most environments will require some form of persistence, if only for audit. Adding policy engine capabilities to a database means fewer overall components. The data held, combined with the database's management features (tables, active rules, views), enables rich and complex representations of state, thus allowing fine-grained policies [29, 24]. Further, databases already operate across a range of components, playing an important role in asynchronous communication, and are designed to be robust, manage simultaneous connections, and can be tuned for performance.

Implementing PE functionality is straightforward when using SBUS-PS (§2.5). All that is required is the definition of active rules (triggers) that represent the policy rules. On a particular event (table update, message receipt), the trigger action can: 1) cause a reconfiguration, by sending a control message (or messages) to the relevant component(s); 2) send general messages to inform component(s) of some change of state; or 3) change the set of active polices (rules).

Though useful for a range of different application scenarios, our initial use of this *database-policy engine* (DBPE) was to manage the components in a physical space/environment.

3.3 Use and conflict

Policy conflict is a serious concern. We have considered issues of conflict detection and resolution for policy that is based on triggers [29, 24], but only concerning conflicts within the same enforcement engine.

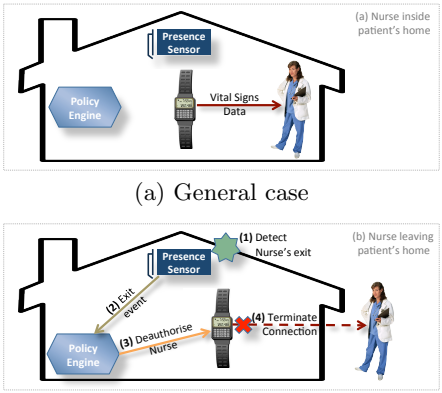
In practice, a number of PEs will operate simultaneously. Thus issues of policy conflict between different PEs is an ongoing challenge. That said, the scope and purpose for which the PE operates can inherently limit the potential for policy conflicts. Firstly, a PE can only execute actions on components for which it is authorised. This means that component owners, or those in control, are able to set the bounds for what is possible. Further, PEs will likely operate within a particular functional scope.

To illustrate, a DBPE might manage a physical space, such that when an individual enters, policy authorises them to discover and access selected local components. This sets up the bounds for, and regulates, the potential interactions between the individual and the environment. The individual's MPE (on their phone) searches and interacts with the components of interest within that (regulated) physical space, controlling the interactions of the phone (the device the individual owns) with components in the environment.

4 Demonstration of approach

This section describes how our policy-enforcing middleware supports assisted living, using real-world scenarios to show how policy enables the realisation of high-level functional goals. We also discuss design decisions relevant to applying policy-enforcing technology in emerging distributed systems.

As the middleware for the PAL project [4], SBUS has been demonstrated to the UK Technology Strategy Board.



(a) General case

(b) Automated response

Figure 7: Carers may only access vital-signs information when physically present

4.1 Assisted living domains

Assisted living involves providing ongoing care and support services to assist those elderly, disabled, infirm or ill.

Given that assisted living is patient-centric, we define a domain for each patient to encapsulate the components pertaining to them. A *domain* refers to a group of resources under common administrative control [28]. Domains are logical constructs, the components of which may, for instance, be in the patient home, some may be mobile (e.g. phones), and others hosted remotely (e.g. in the cloud).

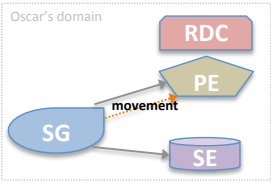
We associate an RDC and a DBPE with each patient domain to manage component visibility, persist data, and enforce policy where appropriate. Most interactions occur within the domain, focusing on the patient and the services they deal with. However, interactions with external entities (e.g. a home nurse, GP surgery) are also managed by the patient's domain, where services are regulated, discovered and accessed by way of the patient's RDC/PE. Fig. 7 illustrates the enforcement of a patient's privacy policy, functionality that is effected independently of the sensor and nurse's applications.

The appropriate structure depends on the environment. Given the trend towards smart cities, it is reasonable to have a domain with a heavyweight PE (e.g. as in §3.2) to govern more fixed infrastructures, e.g. the components in the home, a mall, etc. Other PEs, such as the MPE (§3.1.1), will work to control interactions with the environment. Here, for example, the nurse's MPE automatically connects to the relevant services on entering the patient's home. Our approach enables both centralised and decentralised coordination, as appropriate for the situation.

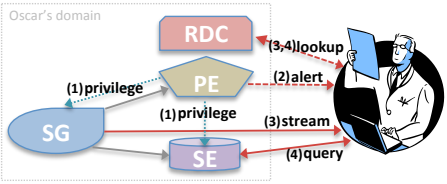
4.2 Scenario: Patient fall—detection and response

We present an implemented scenario demonstrating how middleware capabilities support assisted living. Oscar is an elderly patient who invests in infrastructure to: a) collect detailed data on his daily activities, aiding diagnosis and management of his health conditions, and b) to offer assis-

tance in case of an emergency. His home is fitted with a number of sensors that operate through a sensor gateway (SG). The SG persists data in a storage engine (SE) for subsequent analysis, to provide insight into his well-being. The SG is also connected to a PE to enable a response to significant events. For clarity, we present a conceptual view, in our implementation the DBPE encapsulates both the SE and PE. Fig. 8(a) represents the initial configuration.



(a) General situation



(b) Emergency reconfiguration

Figure 8: Fall scenario reconfiguration

The sensors detect and communicate to the PE that Oscar has collapsed (rapid acceleration/orientation change). The PE maintains policy in order to assess the incident's severity. Here, a rule operates to map the PE to the SG so that the PE receives more detailed movement information (dashed line in Fig. 8(a), rule Fig. 9(a)), and another rule activates a PE detection algorithm on this movement stream to trigger an emergency should Oscar remain motionless (Fig. 9(b)).

```
a) on SG_FALL execute map(PE,movement,SG,movement)
b) on SG_FALL execute load_rule(monitor_movement)
c) on EMERGENCY execute privilege(SG,*,EmServ,Allow)
d) on EMERGENCY execute
      pe_map_send(es_alert,EmServ,*,RDCAdd,@alertparams)
```

Figure 9: Simplified policy rules (only the significant parts shown)

In an emergency, the system reconfigures to enable the Emergency Services (ES) to respond, as shown in Fig. 8(b). Policy operates to alert the ES of the situation, by mapping the PE to the ES, and by sending a message with details of the incident and the location of Oscar's RDC (Fig. 9(d)). The ES are also granted permission to access Oscar's live data from the SG's endpoints, (Fig. 9(c)), and his historical data from the SE. These privileges are reflected in the RDC.

On receiving an alert, the ES operator tries to ascertain Oscar's state, by (manually) mapping to the SG to examine several live data streams, and querying the SE for data prior to the fall. These operations implicitly involve an RDC query. If the ES operator considers the situation serious, an ambulance is assigned and Oscar's streams are diverted (`divert()`) to aid the paramedics response.

This scenario demonstrates the power of policy in driving system functionality. Specifically, it shows that automated policy-enforcement not only effects an immediate response (Fig. 8(b), actions (1,2)), but also makes possible subsequent application/user initiated operations (actions (3,4)).

4.3 Design considerations

Our middleware aims to be open and generic, to support a range of functionality in various environments. As such, we now outline some design considerations, through some micro-benchmarks that indicate the tradeoff between performance and flexibility.

4.3.1 Mapping establishment

First we consider the time to establish a mapping (Fig. 10). Each component ran on a separate Intel Core Duo 2 OSX machine, on an 100BASE-T Ethernet network to avoid the variability of wireless networks. Intuitively, a mapping involving a RDC query takes around twice as long as mapping to a known address, because of the extra connection. This shows the overhead of flexible addressing, which is important in pervasive systems as component availability and addresses are often unknown.

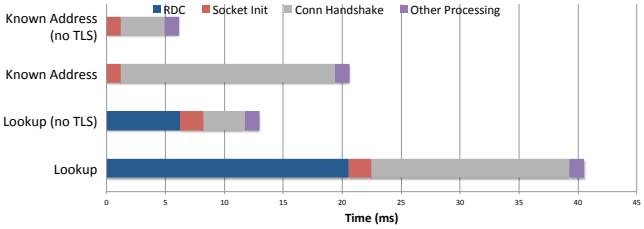


Figure 10: Time for establishing a mapping

We see that the handshaking for a TLS mapping, which validates peer certificates and establishes a secure channel, takes the longest time. This overhead can be avoided in cases where security is a non-issue. To put the times in perspective, it took on average ~22.6ms to transmit 1024 messages (4secs of historical ECG data). This suggests that here the time for even the most secure mapping is not particularly onerous, as catchup is possible even with ECG data, which has one of the highest sampling rates (~3.9ms) of assisted living environments. Whether this suits other application domains depends on the particular requirements.

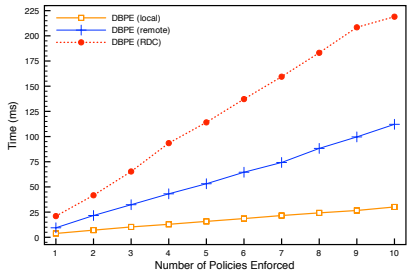
4.3.2 Policy enforcement: Reconfiguration

We now investigate the time taken for the PEs to issue reconfiguration instructions. The experiments involved the PE effecting a number of reconfiguration policies for a particular event, including local components (on the same machine as the PE), remote components (at known addresses), or dynamic component selection through an RDC lookup. The PE was connected to its access point via 802.11g Wi-Fi. The RDC and remote components ran on separate machines. We measured the time to respond to the event (trigger), construct the reconfiguration messages and send them to the relevant components. The results are presented in Fig. 11.

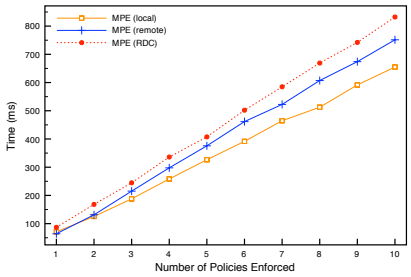
Given policies are enforced sequentially, we see a roughly linear increase in the time for policies enforced.[7] Fig. 11 also indicates the overhead incurred by policies that are dynamic in their addressing, i.e. policy that resolves the applicable components at runtime.

Though the results confirm the intuition that more network traffic entails a greater overhead, this is less pronounced for the MPE. We see similar gradients for all three MPE

[7]Here, each policy was self-contained. If more complex policies are required, e.g. two reconfigurations must happen together, these can be composed (see [24]).



(a) Database Policy Engine (DBPE)



(b) Phone Policy Engine (MPE)

Figure 11: Reconfiguration policy enforcement

experiments, with relatively similar timings, even for the local experiment that avoids network traffic. This is because the overheads of processing, context-switching, etc., of our Android device was more significant than the network traffic. This is compounded by the fact that Android is relatively closed, limiting the ability for customisation and tuning (without rooting), and that we used a Samsung Galaxy S, which in terms of mobile hardware is several generations old (e.g. a single-core CPU). This is in contrast to the DBPE, which ran on a dual-core OSX machine, where the network effects were are more visible. Such factors may be useful when designing for particular environments.

4.3.3 Policy enforcement: Alerting

We also consider a medication reminder (alert) scenario, showing the DBPE processing incoming events, and producing general (non-reconfiguration) messages. An event is sent to the DBPE to trigger the reminder. Policy rules detect the state change and respond by mapping to a component and sending an alert. We measure two types of alerting policy: 1) forwarding the original message; and 2) creating a new alert, which involves a database query (`join`) to modify the message by adding text associated with an identifier (a foreign key relation). We measured the time from sending the initial event to the receipt of the PE-issued alert (similar to a round-trip, but with additional processing). Each component ran on a separate machine, connected by Ethernet, or by Wi-Fi on the same access point—all with the same specifications previously described.

Table 2 presents the mean timings over 500 trials. The Ethernet values are statistically significant ($p < 0.01$); however the Wi-Fi ones are not ($p < 0.29$). This implies that any overhead of the more complex policy action is lost in the overheads and variability of the Wi-Fi infrastructure. As Wi-Fi is the main communication medium for assisted living, it suggests that enforcing policy, even with more involved operations (SQL queries, event creation), does not necessarily introduce a perceivable overhead. That is, much of the speed of the policy enforcement capability is determined by the underlying infrastructure.

Network Connection	Original Message	Generated Alert
Ethernet	16.3490	18.4083
Wi-Fi	30.7355	35.5482

Table 2: Mean reconfiguration timings (ms)

4.3.4 Discussion

We presented measurements to give an indication of the overheads of policy enforcement. This information is relevant in deciding whether such an architecture is appropriate given the operational requirements of the application domain, and the considerations for developing components and policy.

Our numbers illustrate the tradeoff between flexibility and performance. For example, hardcoding or caching location information—which may be suitable for more fixed infrastructure (e.g. components hardwired into buildings)—can avoid discovery overheads, but tends to limit a component to a particular application scope. In pervasive environments, however, it is important that policy can dynamically select the components affected. This is facilitated through RDCs and flexible addressing. Another example concerns security; some data streams are sensitive, and can be protected, at a cost. These are design decisions, depending on the specifics of the application domain and environment. Our results illustrate these overheads. That said, as Table 2 shows, even a less flexible approach (e.g. implementing policy concerns in application-logic, forcing simple policy, etc.) may not necessarily result in an overall performance gain.

At a lower-level, our results highlight the fact performance will depend on the underlying infrastructure. Ultimately, performance will vary according to the implementation and the environment, depending on factors such as the physical infrastructure, operating system, network load, message sizes/frequency, database size and structure, number of rules, users and components, etc. Also relevant is the application domain, and its requirements. It follows that performance results are valid only within the context of the specific deployment.[8]

Overall, our results indicate the feasibility of the policy based approach, and highlight the overheads and tradeoffs of taking particular design decisions. The middleware's subsecond response provides the flexibility for pervasive systems to support a range of application domains.

5 Related work

Our approach to policy-driven middleware complements existing research, and in this section we put our work in the broader context. Importantly, emerging middleware requirements bring challenges [18], and it is unlikely that any single approach could address all of these in their entirety [22].

In terms of *sensor networks* (SNs), wireless sensor networks (WSNs) [1] and body sensor networks (BSNs) [10] are common focus areas. Much of the research examines low-level details of devices, and resource management: power use, communication minimisation, etc. Middleware infras-

[8]We are unaware of any directly comparable middleware, nor standard policy-driven workloads enabling comparison. See [14] for some more general SBUS measurements.

tructure remains at a low level, considering data acquisition, node placement, routing protocols, code deployment and failure handling [32]. More recent work considers the need for virtualisation within sensor networks [17]. Our focus is on providing the broader system infrastructure into which SNs will be integrated, e.g. through gateways. This will include having policy affect SN behaviour, e.g. modifying communication behaviour in emergency situations.

There are many types of reconfigurable middleware. Adaptive middleware [23] allows configuration and customisation, whereas reflective middleware [16] goes further to expose the current system configuration and allow inpection-based reconfiguration. Applications can apply inspection, as can low-level software. Our work complements these types of middleware by working at a higher-level—controlling *across applications*. Our research is based on insight that the *applications themselves* need to be able to be configured and managed, so as to effectively optimise their (inter)operation. This often involves policy representing user-level concerns.

Related to application-level management and reconfiguration is *service composition* (SC) middleware [13]. This involves mapping application-level requests to a set of services [15]. Such work is highly topical given the rapidly increasing numbers of system components [6]. SC is again complementary to our approach, but different from it. SC considers resource allocation and orchestration in a more generic fashion. In contrast, we aim to *directly control* system components and infrastructure, and these controls may or may not be application-specific, and can be for reasons other than service coordination.

Work that involves policy enforcement at a similarly high level [30] often imposes a particular structure/environment. For example, in Ponder2 [31] there are *self-managed cells* and in LGI there are trusted controllers [20]. There may also be restrictions related to a particular form of interaction [28, 33, 19]. In contrast for the emerging middleware applications targeted in this work we try to avoid imposing particular constraints on how applications need to be modelled or designed.

Various models have been proposed that provide complex state representations [7, 5], e.g. to combine and process data from various sources. We are more interested in the mechanisms that enable enforcement and reconfiguration than the contextual modelling itself, although we aim to retain support of developments such as the use of ontologies to effect context modelling [8].

Finally, at the level of interaction paradigms, there is much research into schemes such as remote procedure call (RPC) and pub/sub (see [21] for an overview). The middleware we have developed supports both request/reply and stream-based communication, so as to limit application designs as little as possible. While there has been research into integrating policy directly into pub/sub infrastructure [33, 28, 29], pub/sub is not going to suffice as the only communication paradigm: request-reply is necessary for various control actions, but is cumbersome to implement using pub/sub. Using an event-bus style routing approach has good support for anonymous (or at least decoupled) interactions between components, but this feature may cause security and enforcement difficulties.

Note also that we have recently considered SBUS as a 'proof-of-concept' in exploring the requirements of future healthcare services [26].

6 Conclusion

The major insight from developing middleware to best leverage emerging distributed systems is that *the ability to control, coordinate and manage components is crucial*. Policy plays an important role, since it describes how and when components can and should behave and interact.

To indicate the practicality and technical considerations of policy-enforcing infrastructure, we presented a middleware that uniquely facilitates coordination by providing dynamic, third-party initiated reconfiguration, supports a range of interaction types, and offers a relevant security model. Importantly, such an approach facilitates open, decentralised policy-based control capable of working across components.

The approach and means for enforcement was explored through two policy engines: one integrated into a database, the other for a mobile device. These engines encapsulate user preferences, realising their goals by driving system functionality when and where appropriate. Preliminary results indicate that not only can the policy overhead be manageable, but in certain circumstances, negligible. More experience with real workloads will allow further performance and scalability testing in different environments.

We believe that policy-based middleware is crucial for best leveraging emerging systems, as it allows the use of components for a variety of purposes, and improves flexibility, by abstracting away environmental and usage specifics from application-logic. Note, however, that we aim to support general infrastructure, applicable system-wide. Specialised systems, such as those for high-frequency trading or low-level sensor management are tightly specified, have strict performance requirements, and are tuned to the operating environment. Much is known at design time. Such systems are perhaps better implemented separately, integrated (if appropriate) into the wider systems environment through gateway components.

Currently, we are integrating *Information Flow Control* [3] into SBUS, which involves labelling data in order to track and limit its propagation. This is particularly relevant for cloud computing where application services providers are responsible to their end users yet, with cloud deployment, may use services from several providers, directly and indirectly. Compliance with policy is an unsolved problem for cloud computing. We also have funding to re-engineer SBUS from monolithic middleware to be more modular and lightweight, thus able to bring its reconfiguration capabilities to a variety of platforms and communication infrastructures.

Given technological trends, there must be a movement away from closed, application-specific services to more open environments, where components can be used and reused to meet a range of variable, user-defined functional goals. Our contribution lies not only in making the case for reconfigurable policy-based middleware and in demonstrating that the approach is capable of supporting real-world scenarios, but also in giving insight into the design considerations for emerging distributed systems. We feel that this is an exciting time, as middleware is moving beyond passive 'systems glue', to actively driving system functionality.

7 Acknowledgments

This work was supported by the UK Technology Strategy Board and Engineering and Physical Sciences Research Council, grant TP/AN072C "Personal and Social Communication Services for Health and Lifestyle Monitoring".

8 References

[1] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless Sensor Networks: A Survey. *Computer Networks*, 38(4):393–422, Mar. 2002.

[2] J. Bacon, A. I. Bejan, A. R. Beresford, D. Evans, R. J. Gibbens, and K. Moody. Using Real-Time Road Traffic Data to Evaluate Congestion. In *LNCS 6875*, pages 93–117. Springer, 2011.

[3] J. Bacon, D. Eyers, T. Pasquier, J. Singh, I. Papagiannis, and P. Pietzuch. Information flow control for secure cloud computing. *Transactions on Network and Service Management, Special Issue on Cloud Service Management*, PP(99):1–14, 2014.

[4] J. Bacon, J. Singh, D. Trossen, D.Pavel, A. Bontozoglou, N.Vastardis, K. Yang, S. Pennington, S. Clarke, and G.Jones. Personal and social communication services for health and lifestyle monitoring. In *Global Health 2012*, Venice, Oct 2012.

[5] M. Baldauf, S. Dustdar, and F. Rosenberg. A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing*, 2(4):263–277, June 2007.

[6] A. Ben Hamida, F. Kon, G. Ansaldi Oliva, C. E. M. Dos Santos, J.-P. Lorré, M. Autili, G. De Angelis, A. Zarras, N. Georgantas, V. Issarny, and A. Bertolino. *The Future Internet*, chapter An Integrated Development and Runtime Environment for the Future Internet, pages 81–92. Springer-Verlag, Berlin, Heidelberg, 2012.

[7] C. Bettini, O. Brdiczka, K. Henricksen, J. Indulska, D. Nicklas, A. Ranganathan, and D. Riboni. A survey of context modelling and reasoning techniques. *Pervasive and Mobile Computing*, 6(2):161–180, 2010.

[8] G. S. Blair, A. Bennaceur, N. Georgantas, P. Grace, V. Issarny, V. Nundloll, and M. Paolucci. The Role of Ontologies in Emergent Middleware: Supporting Interoperability in Complex Distributed Systems. In *ACM/IFIP/USENIX Middleware 2011, Springer LNCS 7049*, pages 410–430, 2011.

[9] S. Chakravarthy. Early active database efforts: A capsule summary. *IEEE Transactions on Knowledge and Data Engineering*, 7(6):1008–1010, 1995.

[10] M. Chen, S. Gonzalez, A. Vasilakos, H. Cao, and V. C. Leung. Body Area Networks: A Survey. *Mobile Networks and Applications*, 16(2):171–193, 2011.

[11] T. Dierks and C. Allen. *The TLS Protocol (RFC 2246)*. Internet Engineering Task Force, 1999.

[12] B. Haslhofer and W. Klas. A Survey of Techniques for Achieving Metadata Interoperability. *ACM Computing Surveys*, 42(2):1–37, Mar. 2010.

[13] N. Ibrahim and F. Le Mouël. A Survey on Service Composition Middleware in Pervasive Environments. *International Journal of Computer Science Issues*, 1:1–12, Aug 2009.

[14] D. Ingram. Reconfigurable Middleware for High Availability Sensor Systems. In *ACM 3rd International Conference on Distributed Event-Based Systems (DEBS'09)*, 2009.

[15] S. Kalasapur, M. Kumar, and B. Shirazi. Dynamic Service Composition in Pervasive Computing. *IEEE Transactions on Parallel and Distributed Systems*, 18(7):907–918, 2007.

[16] F. Kon, F. Costa, G. Blair, and R. H. Campbell. The case for reflective middleware. *Communications of the ACM*, 45(6):33–38, Jun 2002.

[17] I. Leontiadis, C. Efstratiou, C. Mascolo, and J. Crowcroft. SenShare: Transforming Sensor Networks into Multi-application Sensing Infrastructures. In *European Conference on Wireless Sensor Networks*, pages 65–81, 2012.

[18] C. Mascolo, L. Capra, and W. Emmerich. *Advanced Lectures on Networking*, chapter Mobile Computing Middleware, pages 20–58. Springer, 2002.

[19] N. Matthys, C. Huygens, D. Hughes, J. Ueyama, S. Michiels, and W. Joosen. Policy-Driven Tailoring of Sensor Networks. In *Springer, Sensor Systems and Software, S-CUBE'10*, pages 20–35, 2010.

[20] N. H. Minsky and V. Ungureanu. Law-governed interaction. *ACM Transactions on Software Engineering Methodologies*, 9(3):273–305, 2000.

[21] G. Mühl, L. Fiege, and P. Pietzuch. *Distributed Event-Based Systems*. Springer-Verlag, New York, 2006.

[22] V. Raychoudhury, J. Cao, M. Kumar, and D. Zhang. Middleware for pervasive computing: A survey. *Perv. Mob. C.*, 9(2):177–200, 4 2013.

[23] S. M. Sadjadi and P. K. McKinley. A survey of adaptive middleware. *Michigan State University Report MSU-CSE-03-35*, 2003.

[24] J. Singh. *Controlling the dissemination and disclosure of healthcare events*. PhD thesis, University of Cambridge, and Computer Laboratory Technical Report TR 770, 2009.

[25] J. Singh and J. Bacon. Governance in patient-centric healthcare. In *i-Society*, pages 502–509, 2010.

[26] J. Singh and J. Bacon. On middleware for emerging health services. In *Journal of Internet Services and Applications (to appear)*, 2014.

[27] J. Singh and J. Bacon. SBUS: A generic, policy-enforcing middleware for open pervasive systems. *University of Cambridge Computer Laboratory Technical Report TR 850*, 2014.

[28] J. Singh, D. M. Eyers, and J. Bacon. Disclosure control in multi-domain publish/subscribe systems. In *ACM 5th International Conference on Distributed Event-Based Systems, DEBS'11*, pages 159–170, 2011.

[29] J. Singh, L. Vargas, J. Bacon, and K. Moody. Policy-Based Information Sharing in Publish/Subscribe Middleware. In *IEEE 9th Symposium on Policy for Distributed Systems and Networks, Policy'08*, pages 137–144, Palisades, NY, USA, June 2008. IEEE Computer Society.

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

[31] K. Twidle, E. Lupu, N. Dulay, and M. Sloman. Ponder2 - A policy environment for autonomous pervasive systems. In *IEEE Symposium on Policy for Distributed Systems and Networks (Policy'08)*, pages 245–246, 2008.

[32] M. Wang, J. Cao, J. Li, and S. K. Das. Middleware for wireless sensor networks: A survey. *Journal of Computing Science and Technology*, 23(3):305–326, 2008.

[33] A. Wun and H.-A. Jacobsen. A Policy Management Framework for Content-Based Publish/Subscribe. In *ACM/IFIP/USENIX Middleware 2007, Springer LNCS 4834*, pages 368–388, 2007.