Technical Report

UCAM-CL-TR-847
ISSN 1476-2986

Number 847

SBUS: a generic policy-enforcing middleware for open pervasive systems

Jatinder Singh, Jean Bacon

February 2014

ISSN 1476-2986

1 Introduction

Computing is fast becoming ubiquitous. The benefits of technology stem from the ability to analyse, process and react to data. As the number of system *components* (data producers/consumers) rapidly increases, the challenge becomes one of management.

This paper presents SBUS, a middleware that addresses this. SBUS was originally developed as a stream-management infrastructure for city-wide transport monitoring [4, 2]. Driven by the concerns of healthcare and lifestyle management [1], it has extensively evolved to enable flexible, high-level policy enforcement.

The requirements arising from these and other application domains could not be met by any existing middleware. First, regarding interaction paradigms, although our systems had a great deal of event-driven functionality, publish/subscribe [5, 6] was useful but insufficient and its paradigm of mutual anonymity not always appropriate (see [9]).

We also needed to support request/response for interacting with services and individuals, and message streams arising from continuous monitoring, for example, of position data transmitted by all the buses in a city. Moreover, we needed an easily-achievable configuration mechanism for a data stream to be sent to multiple services including realtime composite event detection engines, and databases for subsequent analysis of historical data [2]. SBUS therefore supports carefully selected basic communication paradigms above which others such as multicast and publish/subscribe can trivially be built (§2.1).

In emerging distributed systems, it is not just the infrastructure and resources underlying applications that must be configured and managed, but also the applications themselves, in terms of how and when they (inter)operate. Configuration at this higher-level is encapsulated by user-defined policy. Thus the major requirement is the ability to *reconfigure components dynamically in response to changing circumstances*; for example, on detection of a health emergency, a change in location or underlying network, or due to some failure. 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.

SBUS achieves this functionality, uniquely among middleware to our knowledge, by allowing components' interactions to be controlled in a secure, decentralised fashion (§2.4, §2.6). This *coordination* function occurs when *policy engine* components detect specified conditions, which may be as simple as a person changing location or as complex as a distributed composite event pattern. This enables policy, representing high-level concerns, to operate *across applications* to achieve particular functional goals. §4 provides some assisted living examples.

We designed an architecture into which systems and subnetworks can be slotted, see Fig. 1, provided that a *gateway* component can be built to export data outside the system/subnet's environment. In this way, closed or proprietary networks, wireless and body sensor networks and other systems that manage device and resource constraints in specific operating environments, can be incorporated. In practice, closed or proprietary sensor networks usually provide this gateway functionality.

We find that many commercially available systems for environmental and human monitoring operate within closed silos. We believe that our work has created an open system framework for securely reconfigurable components. SBUS has standard middleware attributes, such as a type system, a transfer syntax, secure communication (TLS)

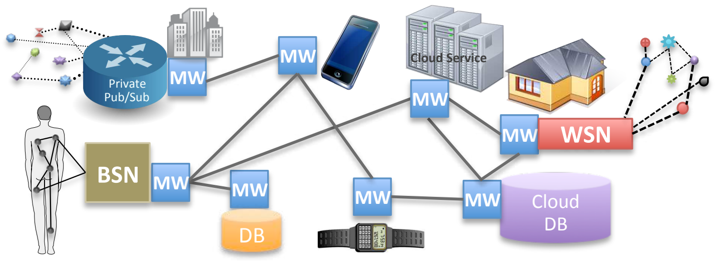


Figure 1: Systems-wide middleware integrating various components and networks.

when required, and resource discovery components. Its main novelty lies in its support for policy-driven reconfiguration of components, with fully decentralised management. Other advanced features are its support for multiple interaction patterns and for endpoint type negotiation. Database components are supported, enabling standard and continuous queries through the common messaging interface (§2.5).

This report is structured as follows: §2 presents SBUS in detail. §3 describes how policy engine components interact with SBUS to effect higher-level goals. §4 presents assisted living use cases and indicates the flexibility/performance tradeoffs. §5 summarises and looks forward to possible future developments.

2 The SBUS middleware

SBUS is a message-oriented middleware for managing interactions in dynamic, pervasive environments. It provides application-independent support, where data and components may be used for a number of purposes.

Crucially, SBUS enables 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.[1] Later we describe how policy engines interact with SBUS to effect higher-level goals.

2.1 General overview

SBUS imposes no particular structure on system design. Its design aim was openness and flexibility, to provide the building blocks to enable any structuring required by the user, applications, or the operating environment.

The basic unit in SBUS is a *component*: an SBUS-enabled process (i.e. an application, service, or part thereof) that uses the middleware to manage its communication. Each component has a number of *endpoints*, which can be thought of as typed communication ports. The endpoints of different components are connected *(mapped)* together to enable communication (Fig. 2).

SBUS supports client/server *and* stream-based interactions. Both may be required in emerging systems. An endpoint takes an *interaction mode*: either a *client* (the query

[1]We build on PIRATES [4], a middleware designed for low-level sensor management for traffic monitoring. SBUS leverages the data representation and communication aspects, providing significant extensions concerning the fine-grained reconfiguration, security and integration capabilities fundamental for enabling high-level, policy-based control.

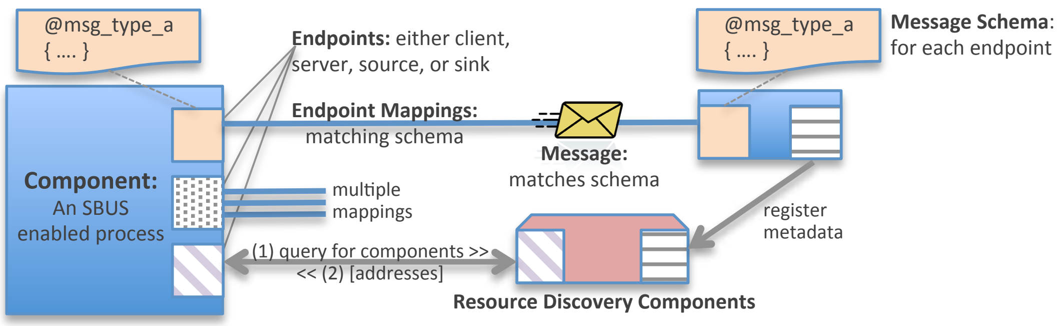

@msg_type_a
{ .... }
Endpoints: either client, server, source, or sink
@msg_type_a
{ .... }
Message Schema: for each endpoint
Endpoint Mappings: matching schema
Message: matches schema
Component: An SBUS enabled process
multiple mappings
register metadata
(1) query for components >>
<< (2) [addresses]
Resource Discovery Components

Figure 2: SBUS Conceptual Overview

issuer) or *server* (returning a result); or for stream communications, either a *source* (producer) or *sink* (consumer). Fig. 3 illustrates the directly supported interaction paradigms. Mappings only occur between corresponding modes, i.e. sources with sinks, or clients with servers. A source endpoint may be mapped to a number of sinks, enabling message multicast.

Communication is naturally peer-to-peer and thus the infrastructure is inherently decentralised. This is deliberate so that more complex interaction models can be built where required. For instance, it is simple to implement pub/sub (event-bus) and message-queue brokers to enable indirect and asynchronous communication. We have developed a pub/sub broker, its core functionality implemented in only a few lines of code.

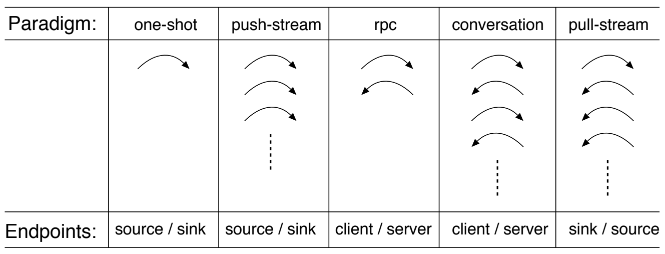


Figure 3: SBUS interaction paradigms

Communication in SBUS is data centric. Data is encapsulated within a *message* of a specific type. A message is often used to capture the details of an event. Message types are described in *LITMUS* [4], allowing expressive definitions, including arrays and lists (Fig. 4). An endpoint is associated with a schema describing the message type(s) it handles; client/server interactions involve a separate type for the query and response. Mappings may only occur between compatible endpoints, i.e. where the type schemata and interaction modes agree.

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 identifiers (hashes) are also encoded to make 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 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, and these can be changed at runtime. They are evaluated in the

`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 4: LITMUS type-representation syntax

context of a message. The language is highly expressive [4]. Filters aid efficiency, by avoiding unnecessary transmissions, and also security, by preventing certain consumers from receiving particular messages.

2.2 Resource discovery

Mappings require the network address of the component with which to connect. *Resource Discovery Components* (RDCs) assist by maintaining a directory of active (registered) components in the environment. A component can register its *metadata*—describing itself, its function, and data handled—with an RDC so that it is discoverable by others. The RDC provides a lookup service, returning the addresses for components whose metadata match the criteria specified in a *map-constraints* query. This enables the runtime discovery of components of interest. Any combination of query criteria is possible. Such constraints tend towards two categories:

Identity: Concerns component specifics, such as its class (named-type), instance-name, author, owner, or public key (i.e. when seeking one specific component).

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

The data constraints enable *schema negotiation.* 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 endpoint on the peer. `Has` ensures that both attribute names and types match; `similar` compares only the attribute types. If agreement is possible, SBUS will enable a connection, and automatically repack incoming messages into the local format (Fig. 5).[2]

There may be any number of RDCs in an operating environment. RDCs may be federated and replicate information, e.g. across a global enterprise. Others may operate within a specific scope, dealing only with a particular set of components, such as those in a patient's house. Any structuring will depend on the application domain. For instance, there may be a number of RDCs in the same environment; several cooperating to manage the components of a large-scale distributed application, and one to handle higher-level services. The location of an RDC must be known/discoverable: perhaps by running at

[2]This is an attempt to balance the advantages of strong typing with the flexibility required for dealing with different environments, e.g. to support mobility (see §3.2).

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

a well-known address; network infrastructure providing the address on connection (e.g. through DHCP options); or by prior knowledge if deliberately obscured. A component maintains a list of RDCs with which it interacts, which is changeable at runtime.

SBUS is decentralised, RDCs exist only to assist. Discovery without RDCs is through *inspection*, where a component is probed to retrieve information via its endpoints and connected peers, enabling service discovery by trawling a connectivity graph. This is useful when an RDC is unavailable, or inappropriate.

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 [4]; although this is not always appropriate. The building blocks are provided for more complex scenarios, such as those requiring information outside component state (e.g. "two A's must be connected to a B"). This allows external components, such as policy engines, to manage the mappings. Such an approach facilitates mobility management (§3.2), e.g. where a person moves between active environments. To assist, 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 the protection and control of middleware operations. These complement application-specific security mechanisms, e.g. system logins or biometric protection for mobile devices. Middleware security falls into three categories:

Transmission Given SBUS is peer-to-peer, control is intuitive because communication is directed. This differs from an event-bus approach where a shared communication channel potentially allows many components to see the same message. To protect the data (messages) and metadata (e.g. protocol state) from eavesdropping at lower network layers, *Transport Layer Security (TLS)* [3] is used. Before any SBUS communication, components exchange certificates, which after validation are used to create a secure communication channel.

Access Control Each component maintains an *access control list (ACL)* for each endpoint describing the components that may connect. When a mapping is initiated, each component examines its ACL to decide if the peer may access the endpoint; the mapping is only established if each peer authorises the other. If privileges change, the mappings are examined to determine whether they remain authorised—if not, the connection is closed.

Currently, access control policy is defined for a component by its class (type), instance name and/or public key. This enables a range of specificity, allowing access control policy to apply to a particular component, or a group. We leverage TLS to provide strong authentication by tying component identity to certificates. This enables verification that a component is who it says it is, and that the access control policy is applied to the correct components. The result is a regulated namespace, which is appropriate, for instance, for assisted living; e.g. if the name of the component should encapsulate a patient-ID, instance names must be governed. If components do not specify a certificate, and thus cannot be authenticated, they may only interact with remote endpoints without access control constraints (world-readable). Of course, the ACL can be extended to incorporate other component metadata or even other authentication systems.

Filtering: There are cases where a particular peer should only receive *some* of the messages emitted from an endpoint. To effect this, filters can be imposed on a mapping to select the messages transmitted. In this way, the filter acts as an authorisation rule evaluated on message content.

2.4.1 Discovery

There will be instances where even the existence of a component may be sensitive. Obviously a component can avoid being discovered, by electing not to register with an RDC. However, this may preclude important interactions. An RDC maintains access control policy to dictate the components that may register and query. However, more flexible, granular controls are also required. As such, we devised an approach where an RDC mirrors the ACLs of its registered components. These are used to filter the results of a discovery query, so that only details of accessible components are delivered.

Discovery by inspection can reveal sensitive information. SBUS provides two forms of control. First, a component maintains access control policy restricting the components that may inspect it. Secondly, a component can dictate whether its existence is revealed to others in an inspection operation.

These measures, though *security by obscurity*, help prevent inadvertent discovery of services, providing an extra hurdle for the malicious. The access control regime still operates to protect the data/metadata even if an address 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 its type, location, the appropriate database driver(s), and *the structure of the data contained.* This

PostgreSQL Table

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

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

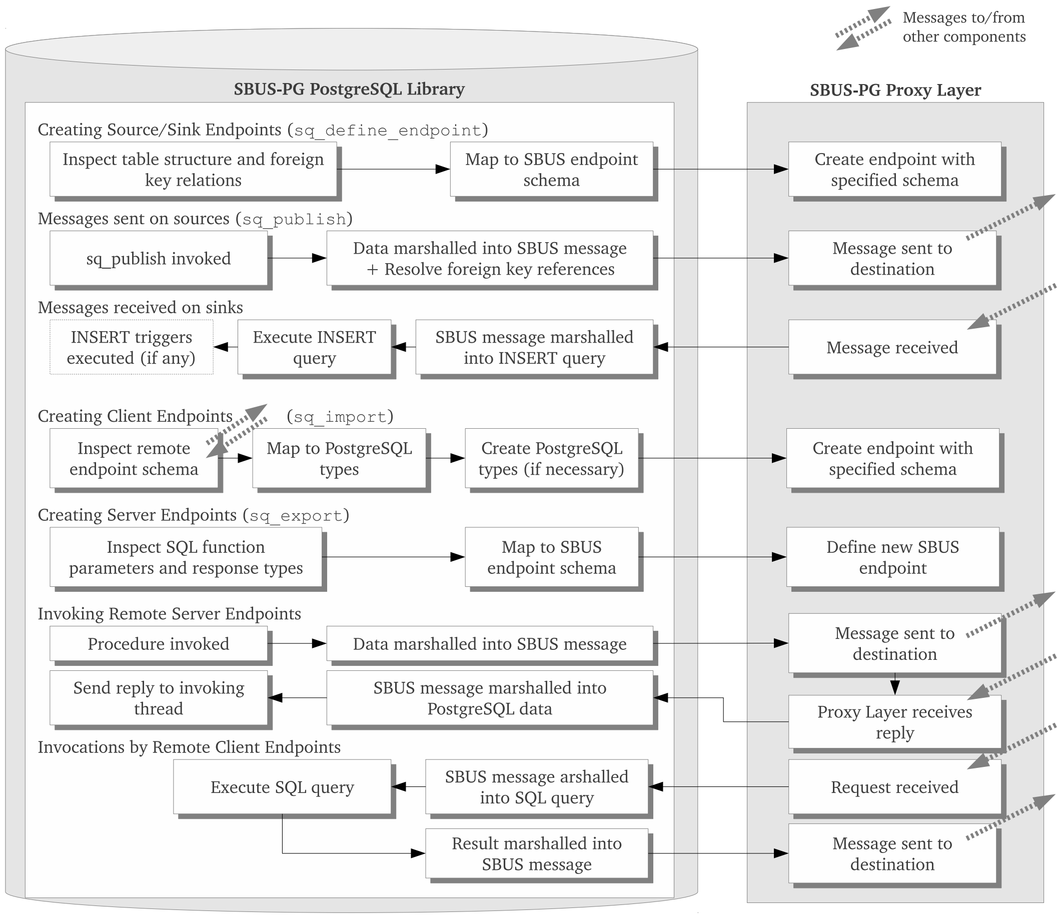
SBUS Type Definition

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

Figure 6: Mapping a table, with a complex type, to a LITMUS schema

is inappropriate for the environment described, where often a component will not know in advance if, when, and where data is persisted, let alone how data is represented in the database, which underpins the ability to query.

SBUS-PG was developed to allow components to take advantage of rich 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. 6). Marshalling between tuples and messages (for SBUS endpoints) is facilitated as the systems are written in `C/C++`. Fig. 7 details SBUS-PG internals.



Messages to/from other components
SBUS-PG PostgreSQL Library
SBUS-PG Proxy Layer
Creating Source/Sink Endpoints (sq_define_endpoint)
Inspect table structure and foreign key relations
Map to SBUS endpoint schema
Create endpoint with specified schema
Messages sent on sources (sq_publish)
sq_publish invoked
Data marshalled into SBUS message + Resolve foreign key references
Message sent to destination
Messages received on sinks
INSERT triggers executed (if any)
Execute INSERT query
SBUS message marshalled into INSERT query
Message received
Creating Client Endpoints (sq_import)
Inspect remote endpoint schema
Map to PostgreSQL types
Create PostgreSQL types (if necessary)
Create endpoint with specified schema
Creating Server Endpoints (sq_export)
Inspect SQL function parameters and response types
Map to SBUS endpoint schema
Define new SBUS endpoint
Invoking Remote Server Endpoints
Procedure invoked
Data marshalled into SBUS message
Message sent to destination
Send reply to invoking thread
SBUS message marshalled into PostgreSQL data
Proxy Layer receives reply
Invocations by Remote Client Endpoints
Execute SQL query
SBUS message arshalled into SQL query
Request received
Result marshalled into SBUS message
Message sent to destination

Figure 7: SBUS-PG library and proxy (component) internals

This integration exposes core database functionality to other SBUS components. `Inserts` involve having relations correspond to a sink endpoint, into which any messages received on that endpoint are inserted. `Selects` are implemented to provide continuous (source/sink) or single query (client/server) functionality. For the former, a source endpoint is also created and linked to a relation, so that subsequent inserts are sent as messages to the mapped components. This is similar to pub/sub. The latter uses stored procedures 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 a server endpoint, so the request message contains the query variables and the response are the messages representing the tuples returned from the procedure. `Updates` and `deletes` are also implemented using stored procedures because the semantics of such operations often require control.[3]

SBUS-PG means that any component can interact with a database without the need to know the relational data structures. The data available to/from the database is immediately visible through the types exposed by the proxy component. While not explicitly part of SBUS, SBUS-PG facilitates components' dynamic interaction with layers of persistence, through a seamless interface (cf. database drivers). This will be important for emerging systems. §3.3 describes how a powerful policy engine was built from SBUS-PG.

2.6 Runtime reconfiguration

Table 1 presents the SBUS API for runtime reconfiguration, which a component uses to change its (and the system) state. SBUS ensures that all related operations are performed, e.g. that removing a privilege closes connections that are no longer authorised, and that the RDC is informed of the privilege change.

`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.

Table 1: SBUS reconfiguration functions

SBUS enables **third-party initiated reconfiguration** (or remote reconfiguration), where a component effectively 'invokes' the SBUS operations of another. This makes it possible to instruct components on how and when to behave; e.g. to map or unmap, update privileges, apply filters, etc.

Such functionality is implemented through *control* messages. Each component has a set of default control endpoints that directly relate to the reconfiguration API (Table 1). If a component receives a control message, it will perform the relevant operation according to the control message's parameters; this is equivalent to self-invocation of the operation. The security mechanisms described above ensure that control messages are only actioned when issued by trusted peers.

Fig. 8 illustrates a component instructing another to undertake a mapping (step 1). This control message forces an RDC query (step 2) by providing lookup query constraints; passing the network address avoids this step. The mapping is then established (step 3).

[3]In practice, it is common for stored procedures to wrap standard SQL operations to allow additional controls and checks, e.g. to protect against SQL injection attacks.

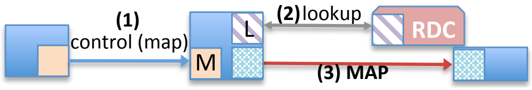


Figure 8: Third-party initiated mapping

It is this capability that enables powerful and flexible policy enforcement. Further, as any component can influence another, it allows decentralised control. This facilitates event-based systems, providing the building blocks to provide a range of functionality across applications and infrastructure.

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. Such functionality is fundamental to enabling policy-driven systems.

3 Policy enforcement

Policy encapsulates a set of concerns, defining the actions to take in particular circumstances to effect an outcome. For middleware, policy traditionally targets issues of network management, resource allocation and/or quality of service, e.g. dealing with node failure, or allocating sufficient resources as requested by an application. Pervasive computing environments, however, also require policy that *coordinates* components with respect to higher-level concerns, dictating how and when they interact. Such coordination, which is external to application logic, enables new functional possibilities as application components can be used/reused in various ways, which may not have been envisaged by the original developers.

This requires the means to control components from *outside their application logic*. As such, middleware is the appropriate point for enforcing such policy.

The third-party reconfiguration abilities of SBUS are crucial to enabling the enforcement of high-level policy, as a component can be instructed by any other (subject to privilege) to perform a particular middleware action. These reconfigurations (Table 1) are 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 enable flexible policy definition. This allows policy to refer to components: a) explicitly, e.g. a particular component, of a certain name or running at a particular address; or b) more generally in terms of desired properties, such as any component in the environment that deals with particular data. This means the components for which policy applies can be determined at runtime, enabling policy that can, for instance, find and connect any data source of type `X` to the local datastore. The appropriate policy, of course, depends on the situation.

3.1 Actions and engines

Often, policies are represented as *event-condition-action* rules, where particular events, in certain situations, trigger an action, e.g. a response in an 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).

From the middleware perspective, policy actions involve:

Reconfiguration: Executing an SBUS reconfiguration operation (Table 1).

Messages: Generating messages to transfer some information, e.g. to raise an alert, inform of some happening.

Policy Management: Policies are contextual, thus a change in state might change the set of active policies, e.g. a set of restrictive privacy policies may be relaxed in a medical emergency, making more components visible to aid response.

It follows that in SBUS, any component can effect policy operations. This is because all components have the ability to send messages, and the potential to reconfigure others.

A *policy engine* (PE) is a service that encapsulates, and enforces, a set of policies. In practice, we expect policy engines to maintain sets of related policies, e.g. as relevant to a particular user, physical space, service contract, and so forth. We have implemented two policy engines, each with a different focus.

3.2 Mobile policy engine (MPE)

The *mobile policy engine* (MPE) is a component built for (Android) mobile devices aimed at supporting mobility, by managing the interactions between the components on the device and its (physical) environment.

An MPE maintains a set of policies (rules) that execute following 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 where the rules apply, enabling fine-grained policy. The MPE was integrated with the *Android Remote Sensing* Service,[4] allowing rules to be defined for any of the 60+ sensor/event streams from an Android device.[5]

The goal is to manage mobility: components will need to adapt as the device moves between different operating environments. At a high-level, policy concerns more than seamlessness, as it is often necessary to change functionality and the associated goals given the new environment. Our focus was therefore on 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.

Mobility motivated the schema negotiation described in §2.2, where components want particular data, but do not necessarily know identity specifics or availability of the components in the new environments.

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

[5]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, external positioning systems, other more general applications, etc.

3.3 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 PE capabilities to a database means fewer overall components. The data held, combined with its management features (tables, active rules, views), enables rich and complex representations of state, thereby allowing fine-grained policies [10, 7]. 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.

Integrating PE functionality was straightforward with 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 action can: 1) cause a reconfiguration, by sending a control message (or messages) to the relevant component(s); 2) send a general message to inform component(s) of some change of state; or 3) change the set of active polices (rules).

We use this *database-policy engine* (DBPE) to manage the components in a physical space/environment.

3.4 Use and conflict

There will be a number of policy engines operating simultaneously within various environments. We have previously considered issues of conflict detection and resolution for trigger-based policy [10, 9, 7]. Issues of conflict arising from competing PEs is a significant challenge, and requires further work. However, in practice, the scope and reach of the PEs can lead to a natural resolution. For instance, a PE can only affect actions on components that authorise it to do so—which, as in the real world, generally means those who own it can control it. This inherently limits the scope for conflict. For instance, a DBPE could be expected to manage the environment, e.g. when an individual enters a physical area, authorising them to discover and access various local components, while the individual's MPE would attempt to search for and interact with the components of interest within that environment (see §4.3).

As this paper focuses on enforcement, we do not discuss policy authoring specifics here. We have previously considered issues of authoring [8, 7].

4 Demonstration of approach

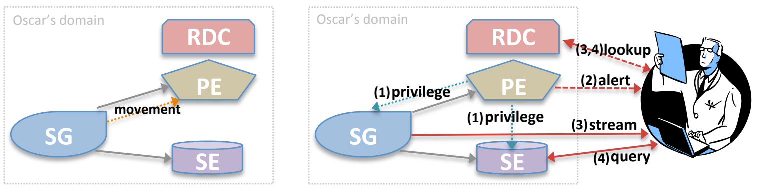
This section presents real-world assisted living scenarios to show how policy enables the realisation of high-level functional goals. We also discuss design decisions relevant for applying policy-enforcing technology in pervasive environments.

4.1 Assisted living

Assisted living entails providing care and support services for individuals, typically those elderly or suffering from particular health conditions. It takes a patient-centric approach to care, where services are tailored to the individual. As the middleware for the PAL project [1], SBUS has been demonstrated in this context to the Technology Strategy Board (UK).

4.2 Scenario: Patient fall—detection and response

We first 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 b) to offer assistance 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 policy engine (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. 9(a) represents the initial configuration.



(a) General situation (b) Emergency reconfiguration

Figure 9: Fall scenario reconfiguration

The sensors detect and communicate to the PE that Oscar has collapsed (rapid acceleration/orientation change). The PE maintains policy to assess the severity of the situation. Here, a rule operates to map the PE to the SG, to receive information of movement (dashed line in Fig. 9(a), rule Fig. 10(a)), and another activates a detection algorithm on the movement stream, to trigger an emergency if he remains motionless (Fig. 10(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 10: A simplified representation of the policy rules (only the significant parts shown)

In an emergency, the system reconfigures to enable the Emergency Services (ES) to respond. This is illustrated in Fig. 9(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. 10(d)). The ES are also granted permission to access Oscar's live data from the SG's endpoints, (Fig. 10(c)), and his historical data from the SE. These are reflected in the RDC.

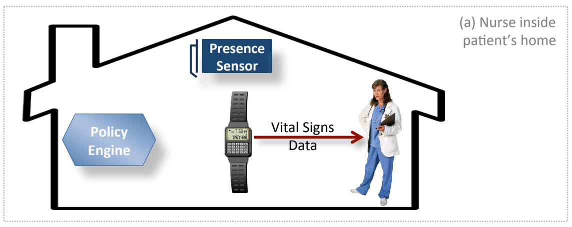
On receiving an alert, the ES operator tries to ascertain Oscar's state, by (manually) mapping to the SG to examine several live data streams, and querying the SE for data prior to the fall. These operations implicitly involve an RDC query. If the situation is classified as serious, an ambulance is assigned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

We use this scenario to demonstrate the power of policy in driving system functionality. Specifically, it shows that automated policy-enforcement not only effects an immediate response, but also makes possible subsequent application/user initiated operations.

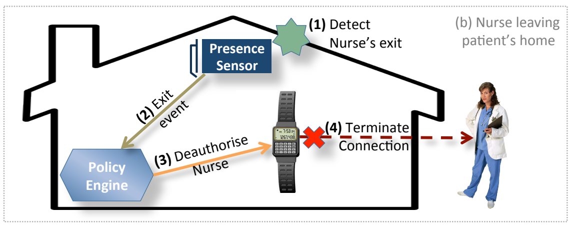
4.3 Domains

A *domain* refers to a group of resources under common administrative control. Domains are logical constructs, the components of which may physically reside, for example, in a patient's home, some may be mobile (e.g. phones), and others hosted in remote/cloud environments.

In line with the patient-centric nature of assisted-living, we define a domain for each patient to encapsulate the components pertaining to them. Each domain has an associated RDC and a DBPE 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. 11 illustrates the enforcement of a patient's privacy policy, functionality that is effected independently from the sensor and nurse's applications.



(a) General case



(b) Automated response

Figure 11: Carers may only access vital-signs information when physically present.

Generally, the appropriate structuring of components depends on the situation and environment. Given the trend towards pervasive computing, it is reasonable to have a domain with a heavyweight PE (e.g. as in §3.3) to govern more fixed infrastructures/ environments, e.g. the components in the home, a mall, etc. Other PEs, such as the MPE (§3.2), will work to control interactions with those within the space (see §3.4). Here, the nurse's MPE automatically connects to the relevant services on entering the patient's home, while the components of the patient's domain regulate the nurse's access to local services. The approach we have taken enables both centralised and decentralised coordination, as appropriate for the situation.

4.4 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, presenting numbers indicating the tradeoff between performance and flexibility.

4.4.1 Mapping establishment

First we consider the time to establish a mapping (Fig. 12). Each component ran on a separate Intel Core Duo 2 OSX machine, on an Ethernet network to reduce variability. 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.

We see that the handshaking for a TLS mapping, which validates peer certificates and

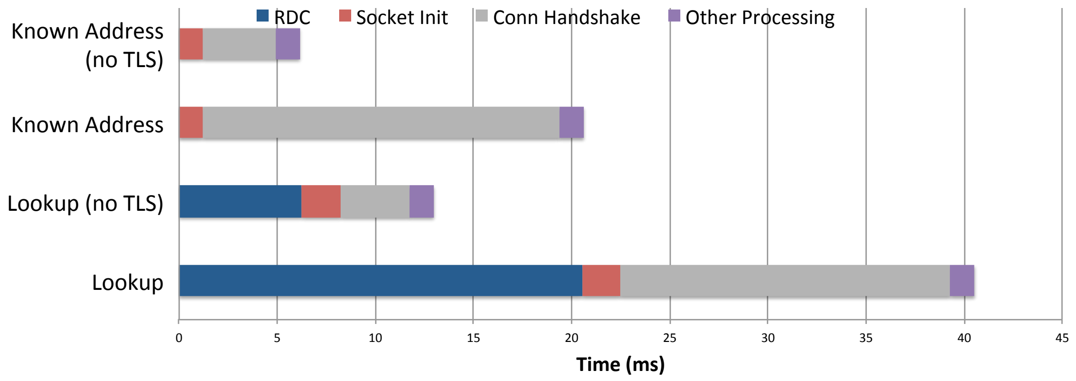
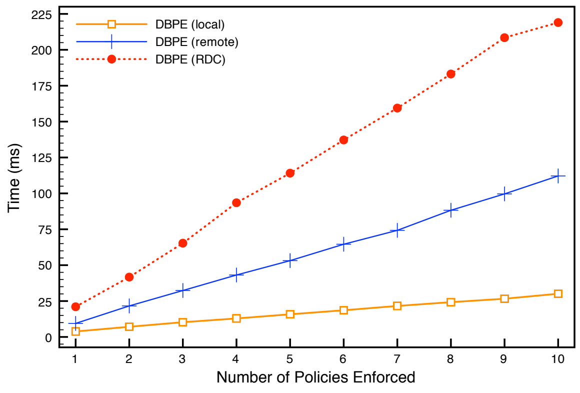


Figure 12: Time for establishing a mapping

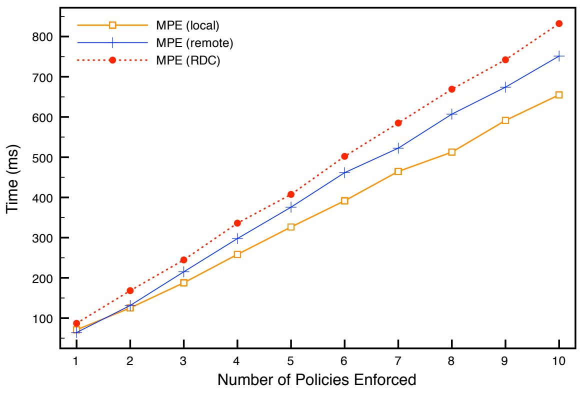
establishes a secure channel, takes the longest time. This overhead can be avoided in cases where security is a non-issue. To put the times in perspective, it takes on average ∼22.6ms to transmit 1024 messages (4secs of historical ECG data). This suggests that here the time of even the most secure mapping 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.

4.4.2 Policy enforcement: Reconfiguration

We now investigate the time for the PEs to issue reconfiguration instructions. The experiments involved the PE effecting a number of reconfiguration policies for a particular event, affecting 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 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 it to the relevant components. The results are presented in Fig. 13.



(a) Database Policy Engine (DBPE)



(b) Phone Policy Engine (MPE)

Figure 13: Reconfiguration policy enforcement

Given policies are enforced sequentially, in a single thread (MPE) or transaction (DBPE), we see a roughly linear increase in the time for policies enforced.[6] Fig. 13

[6]Here we have ensured that each policy is self-contained and thus policies do not interfere which each other. If more complex policies are required, e.g. two reconfigurations must happen together, these can be composed (see [7]).

also indicates the overhead incurred by policies that are dynamic in the 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 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 OS 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 far more powerful dual-core OSX machine, where the network effects were pronounced. Such factors may be useful when designing for particular environments.

4.4.3 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 obtain data used to modify the message by adding text associated with an identifier (foreign key). 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.

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, with a single AP, 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)

4.4.4 Discussion

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

Our numbers illustrate the tradeoff between flexibility and performance. For example, discovery overheads can be avoided by hardcoding or caching location information—which may be suitable for more fixed infrastructure (e.g. components hardwired into buildings)—but tends to limit a component to a particular application scope. However, in pervasive environments, the ability for policy to decide which components it affects, at runtime, is important. 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 a performance gain.

At a lower-level, our results highlight the fact performance will depend on the underlying infrastructure. Ultimately, any timing information will vary according to the implementation and the environment: depending on factors such as the physical infrastructure, OS (e.g. Android), network load, cross-traffic, database size, message sizes/ frequency, 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.[7]

Our results do indicate the practicality of policy-based coordination, and third-party initiated reconfiguration. They also highlight the overheads/tradeoffs of taking particular design decisions. Assisted living, like many real-world application domains that stand to benefit from a pervasive computing infrastructure, operate at human speed. These experiments present subsecond functionality (orders of magnitude faster than humans) which opens up real possibilities.

5 Conclusions and Future Work

The major insight from developing middleware to take full advantage of emerging pervasive computing environments is that *the ability to coordinate components is crucial.* Policy plays an important role, since it describes how and when components can or should interact. We have presented a novel policy-enforcing middleware that facilitates coordination through dynamic, third-party initiated reconfiguration, handles a range of interaction types, and provides security mechanisms. This was demonstrated through two policy engines: one tightly integrated to a database, the other to a mobile device. Our approach enables high-level preferences to drive system functionality, allows the use of components for a variety of purposes, and improves flexibility, by abstracting away environmental specifics from application-logic. Although developed for assisted living, it is generally relevant to pervasive environments.

Our approach aims to be open and general, applying system-wide. It was not designed for, nor suits, specialist systems like control systems, or those for high-frequency trading or low-level sensor management. Such systems, often closed, are highly specified, have strict performance requirements, and are tuned to the operating environment. Much is known at design time. Our focus is flexibility and adaptability, providing infrastructure to meet a wide-range of functional goals, that may not have been previously considered. That said, specialist systems can always be integrated through middleware gateway components (§1).

Our work was part of a collaborative project [1]; our partners considering: usability; health contexts; policy authoring/derivation; sensing platforms, mobility and session continuity. The project has industrial collaborators, aiming at wide-scale deployment.

[7]We are unaware of any directly comparable middleware, nor standard policy-driven workloads enabling comparison. For some general communication numbers, see [4].

Further experience with real workloads will allow further quantified determinations of performance/scalability in particular environments.

We have focused on policy and reconfiguration with respect to interactions. However, there is scope for policy to drive lower-level aspects, e.g. controlling the internals of sensor-networks. While there is existing work regarding quality of service and mobility issues for particular infrastructures, it would be interesting to explore how high-level policy could influence networks generally. This could allow dynamically changing network properties for purely financial reasons, or due to a sudden privacy issue, such as avoiding a nearby individual. Further, while SBUS provides the building blocks for complex state representations, it is worth investigating the application of context models (such as ontologies) to the message types themselves, to aid discovery. Currently, we are integrating *Information Flow Control* mechanisms into SBUS, which involves labelling data in order to track and limit its propagation as it flows through a system.

The current commercial model of technology development favours closed, application-specific systems and infrastructure. Given the directions of emerging distributed systems, there must be a movement away from this to more open environments, where components can be used/reused to meet a range of 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 systems.

Acknowledgements

This work was supported by the UK Technology Strategy Board and the Engineering and Physical Sciences Research Council as part of the PAL project, grant TP/AN072C, 2009-12. We thank Tom Playford for his work on schema negotiation and Daniel Sim for his work on SBUS-PostgreSQL integration and Figures 6 & 7.

References

[1] 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 *Proceedings of the First IARIA International Conference on Global Health Challenges (Global Health 2012)*, Venice, Oct 2012.

[2] Jean Bacon, Andrei Iu Bejan, Alastair R. Beresford, David Evans, Richard J Gibbens, and Ken Moody. Using Real-Time Road Traffic Data to Evaluate Congestion. In *Lecture Notes in Computer Science, LNCS 6875*, pages 93–117. Springer, 2011.

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

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

[5] Peter R. Pietzuch. *Hermes: A Scalable Event-Based Middleware.* PhD thesis, University of Cambridge, and Computer Laboratory Technical Report TR 590, 2004.

[6] Peter R. Pietzuch and Jean Bacon. Hermes: A Distributed Event-Based Middleware Architecture. In *DEBS '02: Proceedings of the 1st International Workshop on Distributed Event-Based Systems*, pages 611–618, 2002.

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

[8] Jatinder Singh and Jean Bacon. Governance in patient-centric healthcare: Let's not forget the providers. In *Information Society (i-Society), 2010 International Conference on*, pages 502–509. IEEE, 2010.

[9] Jatinder Singh, David M. Eyers, and Jean 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.

[10] Jatinder Singh, Luis Vargas, Jean Bacon, and Ken 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.