Reconfigurable Middleware for High Availability Sensor Systems

David Ingram

University of Cambridge Computer Laboratory
15 JJ Thompson Avenue, Cambridge, CB3 0FD, United Kingdom

david.ingram@cl.cam.ac.uk

ABSTRACT

In this paper we consider the problem of sharing data collected by sensors worldwide amongst different applications. We focus on the increasingly large collection of powerful sensor nodes which have full-time connections to the Internet, and describe a distributed architecture capable of flexibly sharing these data streams. The solution must operate reliably and continuously despite live configuration changes and failures. We draw on event-based systems and present the component middleware we have implemented for processing and distributing sensor data. A key aspect is its support for third-party remapping between components in order to adapt to topology changes. We also show that it is able to rapidly reconfigure itself when failures occur.

1. INTRODUCTION

Many pervasive computing applications are made possible by the widespread availability of computation, networking and sensors. Without sensors we can create useful networked systems for humans to interact with directly (such as the web), but with sensors we can go further by observing and controlling the environment.

Embedded networked computers with sensing capabilities have great potential. Application domains include environmental monitoring (pollution, weather, agricultural), security systems, factory automation, supply chain monitoring and smart buildings. In the TIME [1] (Transport Information Monitoring Environment) project we are primarily concerned with road traffic monitoring, though similar design principles apply to other sensor-based systems. To meet the communication needs of large-scale sensor systems in general, we have designed and constructed an event-based middleware, PIRATES.

A key objective of the TIME project is to enable sharing of sensor data amongst different applications. Historically systems have been vertically structured, with physical sensors (which are costly to deploy) attached to dedicated networks (also expensive) supplying a single application run by the organisation which installed them, with a user-base limited to their employees or subscribers only. Within the traffic monitoring domain, isolated applications might show car park occupancy on entry to a city, or display bus arrival times at bus stops. Such data could be used widely both in real time information systems and for historical analysis. If data sharing can be achieved, we can step closer to a planet-wide sensor network, in the same way that the web provides a planet-wide document repository.

A great deal of existing sensor network research targets resource-constrained wireless sensor nets. While this style of system may sometimes be all that is possible, such as in remote areas or hostile environments, realistic applications based on such technology by no means dominate sensor-based systems. For a large and increasing class of applications, including TIME, it is realistic to assume that permanent power and network connections are available (apart from occasional failures). Sensors typically have either a direct connection to the Internet or are represented by a proxy on the Internet which is a single hop away from the sensor. This applies to urban or even rural areas in developed countries. We shall therefore assume that power and network infrastructure covers the nodes in our system.

We also observe the communications convergence torwards IP and the Internet. Our middleware is therefore aimed at data processing within the network, output and dissemination to clients, and also data collection in some cases (for example, mobile phones with attached sensors which typically appear via a gateway as Internet nodes themselves, and have relatively frequent access to a power source).

1.1 Characteristics of the data and network

Sensor data is typically emitted as a continuous stream, arising from periodic sampling at the sensor. Polling and variable rate sampling are less common, due to the simplicity of sensor design (most of which are "black boxes") and the lack of a back channel. Additionally, "pull" models do not work very well when several applications are using the same sensor. Stream handling is therefore very important.

The data streams from sensors are processed in various ways. There may be multiple levels of filtering, for example to clean up noisy data, supply missing information, calculate a statistic or anonymise to respect privacy. Pipes which reformat the data (adaptors) are extremely common, and required whenever two systems with different data formats are connected together.

In addition to data processing pipelines, we may wish to perform sensor fusion, which requires merging different streams, for example to correlate output from two sensors. We also require fan-out connections, for example when distributing streams to large numbers of consumers. There may be different classes of user, such as subscribers, the general public, or emergency services.

Pipeline processing as well as stream merging and fan-out suggest that a pluggable component-based design is required. The dynamic nature of large sensor-based systems means that sensors may be added or removed frequently; it should also be possible to design new applications with previously unanticipated data requirements and connect them into the running system. We must therefore be able to make live topological changes to connections between components.

Stream processing is usually insufficient on its own; most sensor-based systems also employ storage components as well as data mining and triggers. These interactions require an event-based pub-sub (publish-subscribe) or remote method invocation framework.

1.2 Threats to continuous operation

Sensor-based systems incorporate a considerable amount of unreliability, which affects the sensors themselves, the components that process the data, and the links between them. Nodes are subject to temporary outage for hardware upgrades and maintenance, or when machines are rebooted or disks fill up. Individual components are susceptible to crashes, overload or software upgrades for bug or security fixes. Links can be taken down as a result of network failure, configuration changes or refactoring to increase capacity. Finally there are threats associated with data format changes, and staff changes together with lack of documentation.

We wish to construct systems which are reliable in the face of all the above threats. By *reliability* we mean nearly maintenance-free, 24/7 continuous operation. We do not necessarily mean guaranteed delivery of every message (sensor data itself is inherently intermittent).

2. BACKGROUND: EVENT-BASED MIDDLEWARE

There is a large body of existing research into message oriented middleware and event-based systems. Most of this has originally been developed in application domains which do not require sensors, such as stock market systems or news feed aggregation.

Historically CORBA provides remote method invocation, with a language-independent IDL and automatic stub generation.
The CORBA notification service [19] adds a channel-based centralised pub-sub capability, with filtering on the headers of messages. ICE [13, 10] is a cleaner, more modern design with a well developed implementation. This includes a topic-based pub-sub event broker, implemented as a separate server.

Pure pub-sub systems include SCOP, xmlBlaster, Siena and Elvin. SCOP [22] is a centralised topic-based solution for rapid small-scale deployment. Both xmlBlaster [30] and SCOP can also send messages to specific named clients as well as to topics, but a broker intermediary is still used. xmlBlaster uses XML as a message format, with subscriptions in the form of XPath expressions that are evaluated against message headers (the body is opaque). Elvin [23, 24, 7] and Siena [4] both have full content-based subscriptions, but message types are restricted to lists of named values rather than arbitrary structured objects. Elvin is also notable for its comprehensive support of message quenching to reduce the number of messages which are sent. The commercial Elvin product is no longer supported, however.

D-Bus [6], used for desktop integration, gains flexibility by making the broker optional. Point-to-point messages enable RPCs, and broker communications provide pub-sub. D-Bus is a single machine solution, hence cannot be applied to distributed systems, but has the benefit that it is aware of local user identities for access control, and is able to start local services on demand.

Java Message Service [25] (JMS) also has point-to-point as well as pub-sub modes. Messages have expiry times, and are discarded if not consumed before then. The pub-sub mode is topic based, although filtering on the message headers is also possible.

The basic Web Services (WS) stack [27, 28, 16] only provides simple one-way message exchange. The additional WS-Notification specification [17, 18] extends this with brokers for pub-sub interaction. Filtering is based on hierarchical topics, and source quenching is possible.

Hermes [20] and SCRIBE [21] are pub-sub systems layered on top of distributed hash table (DHT) overlay networks, hence solving the resource discovery problem and providing a scalable way to create a network of brokers. Most systems discussed here use a naming service for resource discovery instead. Gryphon [2, 3, 31] and IBM's Websphere MQ [12] also use an overlay network, based on an algorithm for mapping a logical information flow graph onto an existing broker network (although this has to be created by other means since a DHT is not used).

A pure event-based middleware is not required to tackle persistent storage, since it can be provided by applications. ICE ships with a standard storage solution implemented as a separate component and we choose to follow the same approach in our system, since this keeps message transfer and storage as separate concerns. RUNES [5] and SCOP provide some built-in storage capability by means of an auxiliary registry associated with each component. Other middleware designs make storage fundamental to the IPC mechanism itself. For example, ECT [9] and Muddleware [29] are based on data spaces in a similar vein to tuple spaces. Muddleware uses a memory-mapped hierarchical XML database with persistence, queried by XPath expressions but with "watchdogs" (observers) effectively enabling pub-sub behaviour.

IrisNet [8] provides a distributed database for sensor data. It also has similar goals to our project, since it targets internet-connected PC class machines with potentially high bandwidth sensors (such as video cameras), rather than motes with limited resources and simple sensors.

Gryphon is not built on tuple spaces, but makes storage a first class concern nonetheless by supporting event histories (complete logs of messages matching the filter expression) and interpretations (in which a stream of messages is collapsed to some state, such as a derived statistic). An example given by the Gryphon authors is based on car buyers who wish to subscribe to advertisements with certain search parameters. This triggers an initial dump of matching advertisments currently in the database, followed by update events as they occur. The state to be dumped is derived from and not the same as the message history, since sold messages revoke earlier for-sale entries. Gryphon also recognises the need for very common data format changes between components, and hence provides explicit support for event transformations.

Meier and Cahill [14] present a useful taxonomy of event-based systems, although this does not cover all the dimensions we need to discuss in the present paper.

3. ARCHITECTURE

Our middleware solution for constructing robust sensor-based systems is called PIRATES (Peer-to-peer Implementation of Reconfigurable Architecture for Typed Event Streams). The basic PIRATES entity is a *component*. There is a direct relationship between components and processes (i.e. a component is a PIRATES-enabled process), and there may be any number of components running on a given machine. Recall that we shall assume the presence of power and network connections at each node.

Each component has a number of *endpoints*. Endpoints on different components are connected together, or *mapped*. All communication between components takes place via mapped endpoints. The basic mechanism is point-to-point; components send messages to peers directly without requiring an intermediate broker.

The architecture is therefore decentralised, apart from a resource discovery component (RDC), which acts as the name service. The RDC is itself implemented as a component, and there may be more than one, to avoid central points of failure or to create different domains. RDCs may be federated (in which case they exchange state) or separate. RDCs typically run at well-known or easily guessed addresses (such as the standard port number on a local machine).

Most components perform either filtering, merging, storage, distribution or data mining. Figure 1 shows several example components from the road traffic monitoring domain, and a possible set of connections between them.

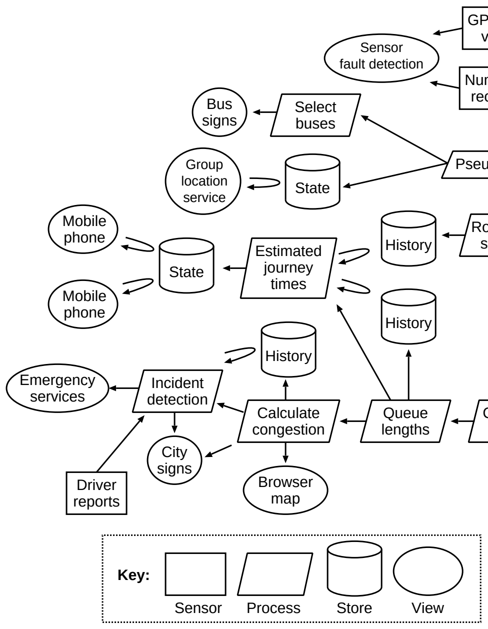

Figure 1: Example components

4. MULTI-MODAL OPERATION

Some kinds of message oriented middleware are more suitable for certain communications paradigms. For example, remote method invocation systems only allow RPC style interactions, whereas Elvin, Gryphon and Muddleware only allow pub-sub (but not RPCs). JMS, MQ and D-Bus allow pub-sub and also messages addressed to specific targets, but no replies. CORBA, ICE, SCOP, Web Services and RUNES are more flexible, supporting RPC as well as pub-sub.

PIRATES extends this idea by attempting to support all reasonable forms of communication between each pair of components. We hope that by doing so a single mechanism will suffice for all of an application's communication needs. Figure 2 presents a partial taxonomy of interaction patterns. We start by observing that in any pairwise interaction one end must send the first message. We assume this is the left-hand peer on the diagram, hence the first arrow is always left-to-right, without loss of generality. There may or may not be a reply. If there is no reply, the interaction is either finished (one-shot), or the originator may continue sending messages (push-stream).

If there is a reply, then again that may conclude business (RPC), there may be many replies (pull-stream) or the process might repeat

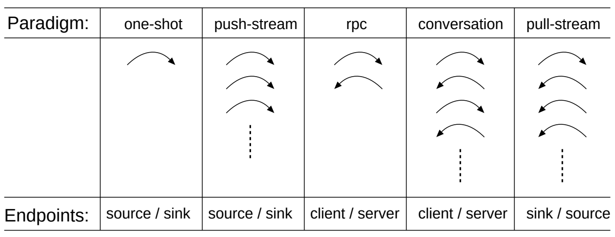

Figure 2: Pairwise interaction patterns

(conversation). More irregular message sequences can be coerced into a conversation by inserting empty acknowledgement messages as appropriate. This therefore covers all the major types of message sequence. Note that a conversation (our name for messages which ping-pong back and forth) is not the same as repeating an RPC interaction many times. In particular, there may be state associated with it. Web applications need cookies because the web effectively provides RPCs, and not conversations.

PIRATES provides all of these interaction types, with the exception of conversations, using four types of endpoint: client, server, source and sink. Clients must be mapped to servers and sources to sinks, but the mapping can be done by either end (e.g. a source may set up a mapping from itself to a sink, or a sink may map itself to the source, and likewise for clients and servers). The source-sink mapping is many-many, and the client-server mapping is many-one. Pub-sub interactions are provided by pull-streams.

Another important distinction is that between a sequence of one-shot messages and a push stream. The latter makes *streams* first-class objects for PIRATES, which is not the case with a normal event broker. The presence of explicit streams makes it possible for tools to understand when two components are "connected", which could not necessarily be determined from a sequence of single events. This allows the stream to be automatically remapped if one of the components moves or terminates, for example.

5. WRAPPERS

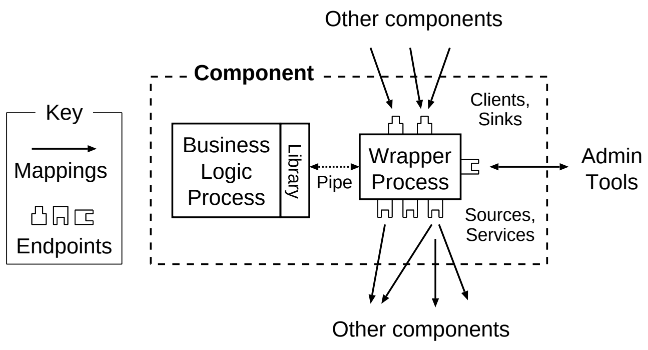

Figure 3: Parts of a component

Figure 3 shows the structure of a single PIRATES component. It consists of the application (business logic) process, which communicates over a local pipe (via library calls) with a *wrapper* process running on the same machine. The wrapper is provided and is the same program for all components. It is implemented in C++, but applications may be written in any language for which there exists a language binding. Bindings are simply ports to the appropriate language of the library which talks to the wrapper. The wrapper is responsible for maintaining all connections to other components.

As well as increasing portability, the wrapper architecture is designed to solve a specific problem; namely to make PIRATES decentralised *and* decoupled. A benefit of a decentralised (peer-to-peer) architecture is that it avoids bottlenecks caused by centralised components in an event broker network. Central event brokers, on the other hand, do provide the important benefit of decoupling producers from consumers. Without this, every component must deal with situations where the peers they are connected to suddenly fail, or perhaps are not running when the component initialises. This leads to tedious error-handling logic in each component, as well as complex dependencies affecting the order in which programs can be started.

Wrappers mitigate this by taking care of all these issues on behalf of their components, just as if there was really a central broker. For example, the business logic does not have to be informed if a producer (source endpoint) they need is not currently running; instead of an error there will simply be no messages of that type until a suitable source is started, at which point the wrapper will connect to it and start delivering messages.

The functionality provided by the wrapper includes accepting subscriptions from other components, maintaining the list of peers interested in a particular event source, performing filtering according to the subscriptions currently in force and delivering notification messages to subscribers. The business logic part does not have to be aware who, if anyone, is listening to the events it emits – hence it can behave just as if there was a central broker.

Effectively, every component automatically includes the pub-sub functionality that is normally provided by brokers, but not the message *aggregation* (pooling messages from different sources, before filtering them via subscriptions). It is still necessary to map directly the components which produce the data you need. For those cases where aggregation is essential, PIRATES ships with a separate broker component. This is a very simple program indeed since the wrappers do most of the work. The broker has one source and one sink endpoint; it accepts connections from anyone to either endpoint, gathers all the messages received on the sink, and sends them out through the source (via the subscription expressions the wrapper has received, of course).

The philosophy we use is to avoid intermediaries (brokers) unless they are actually needed for aggregation. A typical stream of sensor data is more efficiently mapped in a point-to-point manner without a broker – messages which match the filter expression travel one hop instead of two; those which don't match travel no hops instead of one. Latency is also reduced, and the bottleneck component is eliminated. If a single source would become overloaded by clients, additional relay components can provide multiple levels of fan-out.

Many event-based middleware systems, such as SCOP, xmlBlaster, Hermes or Elvin, only allow communication via intermediate brokers. Systems which offer a choice of point-to-point mode vs. brokers include JMS, WS-notification and D-Bus (although they do not provide the decoupling properties we achieve with the wrapper).

Another function provided by wrappers is to implement a collection of *built-in* endpoints. These are endpoints which all components possess automatically. For example, there is a built-in endpoint which returns a list of the other endpoints supported by the component, and another which returns its load statistics.

6. DATA MODEL

The two data models employed by most message oriented middleware are name-value-type triples (attributes), or arbitrary structured types (objects). Siena and Elvin restrict message content to attributes, whereas CORBA, ICE, SCOP and JMS use objects. PIRATES also follows the object data model, because it is richer and maps better onto arbitrary programming language data structures.

PIRATES uses a schema language called LITMUS (Language of Interface Types for Messages in Underlying Streams) to describe message formats. Every endpoint has a schema associated with it to describe the types of message it handles (client and server endpoints also have a second schema for the format of the reply messages). LITMUS is essentially a type system similar to OMG IDL. A summary of LITMUS syntax is shown in Fig. 4, and a sample message format for an endpoint in Fig. 5.

Older versions of our system encoded messages in XML, with RELAX-NG [15] as the schema language. RELAX-NG, like XML Schema [26] language, uses XML itself to express the schemas (unlike DTDs). We prefer RELAX-NG to XML Schema because it is terser and more human-readable. The experience we gained from this earlier system suggested that an XML schema language such as RELAX-NG was more than adequately powerful, but still a little verbose. Secondly, it was so expressive that it was difficult to restrict it to the unambiguous constructs (for mapping onto programming language types) that we wanted. Schema languages are good at describing document formats, but less suitable as type systems. We also found that the default XML message format consumed too much bandwidth.

For these reasons PIRATES now uses a binary encoding natively for inter-component messages, and the LITMUS IDL. LITMUS comes with a rich set of primitive types built-in, including date-time and location. These are useful because many sensor events have an associated timestamp and position in space. PIRATES also supports an XML message encoding, which is in fact used for the library to wrapper link, and can be used to import and export messages from other sources and to disk. Other systems which support both XML and binary message encodings include Elvin, D-Bus, IrisNet and Web Services. SCOP only supports XML and CORBA only supports binary.

`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 syntax

6.1 Marshalling

PIRATES follows the model of SCOP, in which the basic system does not automatically generate stubs with marshalling code from IDL. Instead, library calls are used by the component developer to perform marshalling and unmarshalling (typically the programmer will encapsulate these within a constructor or other method of the object which holds the data). This is less convenient than automatic stub generation but avoids imposing a build system and machine-generated code on every program. A stub generator could be layered on top as an optional tool. In contrast, CORBA and ICE both employ automatic stub generation. RUNES uses OMG IDL but in

```
@fleet
(  taxi
   {  int number
      status < #prebook #hired #forhire >
      txt destination
      clk timestamp
      loc place
      [ bin image ] } )
```

Figure 5: Sample endpoint message format, in LITMUS

the manner that PIRATES uses LITMUS: as a type definition language without automatic stub generation.

6.2 Type checks

A basic RPC service with automatic stub generation might be used to compile both a client and server at the same time, using the same IDL. In that case one could be reasonably confident that the two programs will be compatible, and deploy them without any runtime checks on message format. For any realistic component based system this is not the case, however. Firstly, components may be built by different people, at different times, on different systems. There is no guarantee that they have obtained the right schema, particularly if it evolves and there are different versions in circulation. Secondly, components should not crash if someone deliberately uses the incorrect schema. A consequence of this is that components cannot make assumptions about the format of messages they receive from any other component; we need to check the types.

One approach would be for the middleware to check the basic layout, then hand the problem over to the application. For example, if an attribute data model is used then the application will always receive an associative array, and can consult it to see if required attributes are present. This is tedious for the application writer if there are many things they must check. Instead we would like the middleware to check the entire message first; if this check succeeds the application knows the message contains everything it expects, and can omit further error-handling code.

Type checking is a two-stage process. Firstly, the type of the incoming message must be established. Secondly, it must be determined if this is acceptable to the receiving endpoint. With the attribute data model, the type of the incoming message is explicit. With the object model, some type information is needed however. Reasonable approaches would be to make the object structure fully self-identifying, or to send a single unique composite type identifier. Now, all elements in LITMUS are named, so that programmers can request a data member of a structure by name, such as "height", rather than a position in the structure, e.g. "4th". Consequently sending full type information would add a great deal of overhead; even 1-bit fields would need an associated text string for the name. For this reason we decided not to send full type information.

The problems with unique type identifiers are how to generate them, and how to look them up. If a small probability of collisions is tolerated, they could be generated with a very large random number (perhaps combined with an initial IP address or timestamp), but a repository is then required to store them. It is also highly unsatisfactory that two identical types generated by different people should have different IDs.

PIRATES solves these problems with two mechanisms: LITMUS codes and schema caches. LITMUS codes are a substitute for global type ID numbers. They are 48 bit numbers which are calculated as a hash of the full schema itself. Before hashing, the schema text is parsed to normalise it (removing whitespace and comments, and factoring out dependencies on the order in which sub-types are defined, etc). The resulting number takes up very little space (it is included in every message which is sent) and allows fast type checks, but is stronger than a unique type ID number. In particular, if two LITMUS codes match, then with very high probability the programmers of the two components were working from the same schema. Useful schemas could be published informally for cooperating developers to access using other channels, for example on web pages or in documentation or standards. A component developer includes in their program the LITMUS codes of the types they expect. Since the library only passes validated messages to the application, the developer is then freed from any error checking: if the code represents a schema containing a member *foo*, they can manipulate the member *foo* in the message without any concern that it might not exist.

Some components, such as the event broker or a bridge, must accept messages of *any* type. These are called polymorphic endpoints. These will be situations in which a polymorphic endpoint receives a LITMUS code it does not recognise. When this happens, the receiving wrapper suspends processing of that message and calls back to a built-in endpoint called `lookup_schema` on the sending component, passing the unknown LITMUS code in question. The other component looks this up in its schema cache, which is maintained by the wrapper, and replies with the full text of the associated schema. This is immediately added to the cache of the receiving component, to avoid repeated lookups if more messages of this type are sent in the future. Since all components do this, schemas pass along chains of polymorphic components. The invariant is that no component will ever transmit a message it does not know the full schema for. This makes central schema repositories largely unnecessary. It is also possible to create polymorphic source endpoints and to define new types dynamically.

6.3 Schema evolution

A common characteristic of sensor data formats is that they change over time. Often new fields need to be added; in the course of our project we have also encountered situations where the semantics of existing fields change, such as the introduction of an error value. In the former case it is highly desirable that existing data consumers continue to function with the stream containing the extra field[s]. If this is not the case then a great deal of infrastructure must be updated (probably not all under our control) when we change the message format. Even if this can be achieved, we must either halt the data source temporarily or first upgrade the sinks to deal with both the old and the new message types, neither of which is particularly attractive. ICE has some support for this with its *facets* mechanism; for example a component may simultaneously support two facets of an interface, one to deal with the old message type and one for the new. PIRATES also allows a component to have multiple endpoints with the same name but different message schemas, however this is not our method of choice for schema evolution or component versioning.

The attribute data model neatly avoids the schema evolution problem, since new fields will simply never be requested by applications and hence are effectively ignored without disrupting the old fields. Elvin is an example of a system which ignores non-addressed attributes in this way. Attributes also make it easy to write filter components which look for a few key values in the message only, then distribute the entire message to other components which will understand the full type (for example, we might wish to write a generic filter that selects messages of any type with timestamps in a certain range, or location within a given neighbourhood, and forward them to a dedicated regional server for processing).

The object data model (which we use) is more expressive than attributes, but does not lend itself so well to schema evolution. Most of the systems with the object data model which support schema evolution at all do so by means of a type hierarchy. Examples are ECT and Hermes. This allows fields to be added by sub-typing the original message format. The disadvantage of this is that an explicit type hierarchy relation must be defined and distributed.

PIRATES is designed to allow schema evolution in future, although this is not yet implemented. The design works by relaxing LITMUS code checking in two ways, called partial matching and sub-tree matching. For example with partial matching, an unrecognised LITMUS code would trigger a request for the full schema in the normal way. Once received, the schema can be examined to see if it *covers* the expected schema. One schema covers another if it has the same root, and when overlaid on the other tree it replicates it apart from an arbitrary number of extra branches (at any node). If this is the case then the type would be marked as compatible in the component's schema cache, and the pruned version delivered to the application.

6.4 Event filtering

Pub-sub middleware allow subscription expressions of different generality. The simplest is channel-based (this used to be the CORBA model). The next most general is topic-based subscription. This is used by ICE, SCOP, WS-notification and SCRIBE. Going further we may allow filtering on the message content, using special header fields for this purpose; this is the approach taken by xmlBlaster and newer versions of CORBA. The header is like a dictionary attached to the main message. JMS allows topic-based as well as content-header subscription. Finally filtering may be done on the full message body. This is the technique used by PIRATES, and also by Gryphon, MQ, D-Bus, Siena and Elvin (although the latter two employ the attribute data model, hence the message body is effectively a header itself). PIRATES and MQ both support topics as well as content-body subscriptions. Figure 6 shows a content-based subscription in the expression syntax used by PIRATES. There can only be one subscription expression per mapping between endpoints, but this can be changed at any time. Third parties can also change subscriptions using a `resubscribe` built-in endpoint.

```
vehicle? & vehicle/timestamp > 'startdate' &
vehicle/timestamp < 'enddate' &
vehicle/speed > '50' &
(vehicle/description/colour = '*green' |
(vehicle/occupants.items = 1 &
 vehicle/occupants/#1 = 'alice'))
```

Figure 6: Subscription example

7. NAMING

During early PIRATES development, the desire for unambiguous component names led to some very awkward naming, such as `filtered-average-fused-local-bus-data` or `daves-taxi-component`. Of course unique names could be guaranteed by appending the component developer's e-mail address, but these would not be very long lived if the developer for a component changed. To avoid these problems, we do not require component names to be unique at all, and instead allow a broader set of criteria with which to describe and select components.

When a component needs to locate another in order to map an endpoint, it sends a *map constraints* string to the resource discovery component. This includes fields for component name, author, endpoint name, endpoint type, LITMUS code, keywords, public key, and component ancestors. All fields are optional apart from endpoint type and LITMUS code, which are always used to ensure the component mapped to is type compatible. The public key is the only way to *guarantee* that a certain component is selected. Component ancestors are third-parties which a given component is currently mapped to. This provides a way to request a component which is connected to specific others. For example, if there are two standard filter components with the name `pseudonymise`, it is possible to request the one connected to bus data instead of the one connected to a car data stream, should pseudonymised bus events be required. RDCs always reply with a list of all the components they know about which fulfill the requirements expressed by the map constraints string.

8. THIRD-PARTY REMAPPING

A large, continuously operating sensor data processing system will have a frequently changing configuration. To support reconfiguration of the running system, PIRATES provides *third-party-remapping*. The mechanism for this is based on three built-in endpoints: `map`, `unmap` and `divert`. The `map` endpoint accepts requests to remap the component's endpoints (change which other component[s] they are connected to), and performs this without involving the business logic itself. This inversion of control makes it possible for management tools to effect topology changes on live components, with no special support from applications. For example, if a component needs to be migrated (to a machine with more resources, for example), or we wish to switch to a different supplier for some events, this can be done by remapping the endpoints using the built-in interface. Tools which employ this technique may be interactive (such as a GUI manager's control screen, for setting up maps between components visually using drag-and-drop) or fully automated (such as a load balancing tool which moves components to different machines as required).

The `divert` endpoint performs several remaps in one go: it instructs every other component connected to a particular endpoint to remap to an alternative. A component can therefore be moved easily by starting a new version elsewhere, then calling `divert` on all the original's endpoints to switch any number of clients over to the new location.

Soon after implementing this we observed that *updating* a component in place requires exactly the same mechanism as component migration. Often we may produce a new version of a component with exactly the same interface as before, but fixing some minor bug, adding features unrelated to IPC, or with increased performance etc. In this case the third-party remapping is used to substitute the running component with the updated one (typically running at a different port on the same machine) without having to stop the component and with no interruption to service for other components which depend upon it.

RUNES [5] (Reconfigurable, Ubiquitous, Networked Embedded Systems) is an existing system which is most similar to our approach, as it also offers components with dynamic reconfiguration for the interconnections. Endpoints in RUNES are called *interfaces* or *receptacles* (an interface must be connected to a receptacle). OMG IDL is used for interface types, but without CORBA-like stub generation; our LITMUS-based approach is very similar in this respect. The critical differences occur because RUNES is targetted at embedded systems. RUNES components may be lightweight, in which case they can be implemented as passive entities (some code in a chip on a small device) without a separate process. All PIRATES components are active, so they can respond to external requests on the built-in endpoints, with the important benefits of third-party control and reflection. This gives a more uniform component model.

ECT is another system which allows third-party reconnection, albeit within a centralised architecture. Gryphon and MQ also make connections for you based on an abstract information flow graph created by the user. RUNES and Gryphon additionally support *interceptors* (lightweight filter components), which are transformations that can be inserted into streams at component interfaces by third-parties. These may be used for encryption, compression, debugging and gathering statistics etc.

8.1 Automatic reconnection

An important benefit of the wrapper architecture is that when a connection to another component is lost, the wrapper may attempt to failover to an alternative replica transparently, or wait until the other component is restarted. The application believes that it has an uninterrupted connection. In order to achieve this decoupling, the wrapper must know the correct thing to do without consulting the business logic. In simple cases this can be achieved if the wrapper re-issues a map request (using the original map constraints string) to the RDC. If there is no state associated with ongoing interactions then any component which satisfies the constraints will do (were this not the case then the map constraints must have been underspecified).

More complex reconnection logic (such as “there must be at least two A’s connected to every B”) is best implemented by an external agent, since it requires non-local information. We call such an agent a *mapping engine*, and it effectively lifts reconnection policy out of the components’ business logic. Wrappers simply report to the mapping engine when they have lost a connection. Currently the mapping engine service is also provided by the RDCs, and hence is also distributed and federated. The problem now is that connection policy is present in two places: the initial maps are performed procedurally by the component’s business logic, and any subsequent remapping is performed by the mapping engine.

Our experience with the system has suggested that the most elegant solution is for almost all components to *start unmapped*. That is to say, although the library still provides mapping calls to the application, we choose not to use them; all mapping is done via the third-party remapping endpoints by the mapping engine. Applications start in an inert state (running, but not receiving any events due to their unmapped endpoints). When they register their presence with a RDC, the mapping engine is triggered to perform initial maps of their endpoints.

The benefits of this arrangement are that all connection policy is held in one place (the mapping engine’s tables), and none of it is hardcoded into applications. A distributed application comprises the executable components, plus mapping rules for connecting them together. Programs are much more likely to survive configuration changes if they do not contain explicit component names or map constraint strings.

Mapping engines therefore offer significant advantages over traditional styles of network programming, as shown in Figure 7.

8.2 State migration

During component migration, stateful components must recreate their necessary internal state. Our model assumes that the state of each component which needs to be preserved is purely a function of the messages it has received. State migration is the responsibility of application logic. Each endpoint emits messages tagged with sequence numbers so that state checkpoints can be named. The recent message history can be replayed from any endpoint’s circular output buffer, provided the configurable message buffer space has not been exceeded in the time taken to restart the component. The existence of explicit mappings between components together with sequence numbers allows streams of sensor data to be correctly reconnected, which would not be possible with an anonymous publish-subscribe system.

1. Message-based	`conn = connect(peer)`
	`if conn.failed() error()`
Time coupling	`conn.send(message)`
Hardcoded peer	`conn.close()`
2. Event-based	`publish(topic, message)`
Time decoupled	
Peers must agree topic	
3. Mapping engine	`endpt = declare(localname)`
	`endpt.send(message)`
Time decoupled	———
No hardcoding	*Mapping rules:* { (from, to), ... }

Figure 7: Comparison of address specification

9. MAINTAINABILITY

Systems which rely on central brokers are relatively easy to maintain, because the broker can be queried to discover which clients are running. Maintenance is trickier for decentralised systems. In keeping with our philosophy of providing the benefit of a centralised architecture in a peer-to-peer system, PIRATES provides two built-in endpoints to make maintenance easier: `get_metadata` and `get_status`. Anyone can call these given a component’s address in order to interrogate it.

The `get_metadata` endpoint returns the component’s name, author, description, keywords and public key. It also lists the endpoints provided, and for each one provides its name, type, and LITMUS codes for message and reply types. These can of course be converted to full schemas with `lookup_schema` if necessary. All of this forms the static metadata associated with the component. Self-describing components are good for humans, as well as automated tools such as mapping GUIs.

The `get_status` endpoint returns the dynamic status of the component. This includes its address, creator (the user who instantiated it), instance name, load and latency. For each endpoint, information is also returned on number of messages processed/dropped, the subscription (if one is enabled), and the list of peers that endpoint is currently connected to. Furthermore for each peer a structure is returned containing the peer component and instance names, endpoint, address, remote subscription and latency (this information is known to the wrapper, as it is exchanged during the handshake which takes place when an endpoint is mapped). RDCs call `get_status` periodically on all the components which have registered with them, as a liveness check.

The list of peers is particularly useful, because it allows tools to crawl the connection graph, like a web spider. In this way they can in principle discover the topology of the entire distributed system (of course this is not possible to do precisely because it may change during the sweep). This is the distributed analog of requesting a list of clients from a centralised broker. It is useful for giving a management overview of the network, and also for finding components which are not registered with any of the local RDCs.

10. IMPLEMENTATION

10.1 Concurrency model

The wrapper must process many connections simultaneously and without blocking as a result of network delays. Our implementation uses a single-threaded wrapper which performs all I/O in non-blocking mode, and never blocks. It employs a state machine and an abstract message structure to record partially processed messages. We believe this approach is less susceptible to concurrency problems than a multi-threaded wrapper.

10.2 Integration

Applications typically employ a number of different libraries (for example, to present a GUI), so the PIRATES library must coexist with others. Any library which delivers events has several choices for integration with the program's event loop: it may be passive (so the application can manage the event loop), capture the main thread of control for itself, use callbacks or create new threads or processes. Clearly capturing the main thread of control is not a very polite solution, and callbacks lead to serious concurrency problems. SCOP is passive and hence very unobtrusive, however PIRATES needs to respond to requests on built-in endpoints without help from the application, hence must have an active thread of control.

The solution we have adopted is to make the wrapper a separate process, and integrate the library closely with native programming language I/O. All the library calls in the API use a blocking mode of operation, because that is less error-prone for developers than callbacks. In the C++ binding, the file descriptors used to communicate with each endpoint are exposed so that the programmer can add these to an existing `select` loop, and hence avoid blocking as well as merge PIRATES with other network and file-based I/O. Alternatively, developers may write multi-threaded components. Concurrency control is unnecessary provided threads manipulate different endpoints; each endpoint has a dedicated pipe to the wrapper and the wrapper itself does not block. New endpoints may be created whilst the component is running, so a server may make clones of an endpoint for use by worker threads.

10.3 API

Our primary concern for the implementation, and indeed the design, has been to create an exceptionally simple API. This was the approach taken by our earlier system, SCOP, and also by Elvin; in both cases the libraries have been widely used as a result. By contrast CORBA has rather complicated APIs. A middleware is not well served by a complex API, since programmers need to concentrate on their business logic and not the glue code. The basic list of primitives provided by the library consists of: `start`, `subscribe`, `map`, `emit`, `rpc`, `rcv`, `reply`, `unmap`, `ismapped` and `stop`. Each of these take only a few parameters. For example, Fig. 8 shows complete, compilable code for an event broker component (the actual broker we use is marginally more complex as it also includes configuration code to change the port number).

```
#include <pirates.h>
void mainloop()
{
  scomponent *com; smessage *msg;
  sendpoint *pub_ep, *out_ep; snode *sn;
  const char *code = "FFFFFFFFFFFF";

  com = new scomponent("broker");
  pub_ep = com->add_endpoint("publish",
    EndpointSink, code);
  out_ep = com->add_endpoint("notify",
    EndpointSource, code);
  com->start("broker.cpt");

  while(1)
  {
    msg = pub_ep->rcv();
    sn = msg->tree;
    printf("Received tree:\n"); sn->dump();
    out_ep->emit(sn, NULL, msg->hc);
    delete msg;
  }
}
```

Figure 8: Component code example: Broker

10.4 Portability

In common with other middleware such as ICE, RUNES, xmlBlaster and D-Bus, PIRATES is designed for easy portability to different languages. Its native language is C/C++, since most other languages are higher level and hence easier to port to. SCOP achieves easy portability via a thin language-specific library, and a larger event broker which is written once in C++. With PIRATES we wanted to achieve the same effect, hence the wrapper was implemented as a process rather than a thread (if it were a thread it would have to be ported to different languages, but as a process only the library must be converted).

Our experience with SCOP suggested that even with a thin library layer, keeping the different language bindings synchronised was a lot of work. They were easy to port initially, but the inevitable change requests often did not get applied at once to all bindings, due to the difficulty of switching languages in the middle of a task. For this reason we wanted the PIRATES library to be even thinner, if possible. Unfortunately, PIRATES includes a rich type system, which SCOP does not; the library would have to understand this, and hence we realised would not be thin at all. For example, in order to encode messages for transmission across the pipe to the wrapper, the library would require marshalling and unmarshalling code, and an appropriate LITMUS schema, which relies in turn on the LITMUS parsing code.

Our solution was to defer all type checking and schema processing to the wrapper. The messages sent across the pipe are not validated, and hence cannot be transmitted properly using the compact binary encoding. Instead, we use the XML import/export facilities. The extra bandwidth is not a major problem because the library to wrapper connection is just a local pipe and does not cross the network. Constructs which are syntactically identical in XML, for example lists and structures, or missing optional elements, are disambiguated on receipt by the wrapper, which does have access to the proper schema.

10.5 Modularity

The PIRATES component wrapper is a considerable piece of software (15,000 lines of code) upon which all components depend. Consequently it is important to consider which features are essential to the core middleware, and which can be separated. Modularity also allows pieces of the system to be replaced by alternative versions.

We have two mechanisms at our disposal to achieve modularity: layering and components (vertical vs horizontal separation). For example, PIRATES uses a separate layer for its marshalling code.

Splitting functionality into separate components provides excellent isolation since it runs in a different process. PIRATES uses separate components for the RDC and the Event Broker. The RDC program could be replaced, which allows for a pluggable name service. Message persistence is also pluggable, since PIRATES defers this to other components. Finally, we have made the topology connection logic (automatic component start, automatic reconnection, failover, migration, replicas and load balancing) pluggable with mapping engines. Currently the mapping engine is supplied by the RDC, but this could be separated if required (it is also possible to ignore the built-in mapping engine by not requesting any persistent components or persistent maps, and to implement one's own mapping service instead, since the built-in remapping endpoints are publicly accessible).

11. EVALUATION

11.1 TIME project deployment

Within the TIME project we are using PIRATES to distribute data from a number of sensor types. The length of queues at the traffic lights at major junctions around the city is reported by a SCOOT[11] system, and the position of buses is tracked with GPS units mounted on each roof. Weather conditions are measured by our own weather station and pollution data is fetched from a number of sensors around the city. We also monitor the occupancy of the city centre car parks. To supplement the SCOOT data on one major road we are using an infra-red sensor attached to a lamppost, which counts vehicles from their heat signatures. We are also experimenting with acoustic sensors, automatic numberplate recognition (ANPR) and a data feed from the railway station ticket barriers.

All of the data sources we have used fit the model of direct connection to the Internet, and each source is represented by one PIRATES component. For example, aggregation of the SCOOT data from the 37 monitored junctions around the city takes place within the SCOOT hardware before it is collected by PIRATES. Individual sensors are prone to failure as well as the aggregate stream itself, so the SCOOT data is discontinuous.

We decided to store all sensor data for future processing. Storage is handled by a generic stream persistence component, `spersist`, which can be attached to any stream. It creates an event history in an SQL database on disk. Each event is logged together with its timestamp and associated LITMUS code, so that they are self-identifying if the schema changes during the observed history. A second database table stores all observed schemas with their corresponding LITMUS codes. `spersist` provides an RPC endpoint for replaying events between a specified pair of timestamps.

Output is directed both to periodically generated static webpages and interactive Java applets, which provide statistics to illustrate the city's current and historical traffic states. We have made use of aggregation in a number of ways, for example comparing SCOOT data with the bus GPS traces to predict waiting times at red lights, and using the infra-red camera to validate the other data.

11.2 Design choices

We now consider the major design choices made during creation of the middleware, and evaluate their success (or otherwise).

The decision to use an object data model rather than tuples allows for rich data expression, at the expense of more difficult schema evolution and partial message processing (for example, it is harder to extract and filter on timestamps buried inside a complex object).

We choose to integrate RPC and event style communication, which has made the middleware significantly more complicated, as their needs inside the implementation are somewhat different. The benefit of doing this is that it avoids the need to employ two separate middleware for many types of application.

PIRATES has only one choice of type system. Simpler types of data such as binary, plain text or name-value pairs can trivially be represented within the rich type system using appropriate schemas, hence a separate "content type" field for messages is unnecessary. In retrospect not making the type system entirely optional (as in SCOP) was a mistake, since there is no way to avoid paying the complexity cost for components which do not need it.

Another weakness of the current implementation is the lack of a pluggable transport layer (only TCP is offered). Many systems, such as xmlBlaster, ICE, Twisted, Siena, WS-notification and Elvin allow a choice of transport (TCP, UDP, HTTP, e-mail etc). Our experience suggests that more attention should be paid to modularity in the future.

Allowing subscriptions to filter on the full content of messages seems a fully justified design choice. Point to point communication has also improved latency and eliminated bottlenecks associated with unnecessary event brokers.

The lack of any built-in topology setup has ensured that PIRATES is policy free, at the expense of a reliance on RDCs. In the future we envisage the equivalent of search engines, using the topology discovery built-in endpoints in order to explore the component space. This could add the same ease of use to a relatively unstructured distributed sensor system as conventional search engines do for the web.

A success at the implementation level was the non-blocking (rather than multi-threaded) approach to concurrency. We also made two notable mistakes. The first was choosing C++ as the native language rather than Java; this has made the wrapper unnecessarily complex. The second was making the wrapper a process rather than a thread. The benefit of increased portability to different language bindings was probably outweighed by the additional complexity and overhead added by the wrapper to library communication protocol.

The most powerful design choice we have made was the decision to allow third-party reconfiguration of the inter-component mappings. This has been very successful and instrumental in allowing the middleware to adapt to the changing requirements of the project.

11.3 Performance

The scalability of PIRATES depends on the topology of the connections between components, which is not mandated by the middleware itself. Higher level policies are expected to avoid bottlenecks and instantiate a suitable number of broker components if necessary. The performance of the middleware itself must however be predictable in order for such capacity planning to take place.

Component performance tests were carried out on a live system running on a network of 1.8 Ghz Transtec SENYO 600 mini-PCs using Linux. Figure 9 shows the round trip time (RTT) of RPCs with payload sizes of 100 bytes or 10 Kbytes, as the load on the machine increases. The load consists of between 0 and 20 parallel components, each sending 100 messages per second. For small message sizes the RTT starts at 2ms and is not greatly effected by load, rising to a maximum of 5ms. For larger messages the RTT starts at 45ms, rises as the load on the machine increases due to contention for the CPU, then falls back almost to the same value as the OS scheduler starts to prioritise the measured component in favour of the heavy background load.

Figure 10 shows the round trip time of RPCs in the presence of crosstalk messages: in this case between 0 and 20 other components are each sending 100 messages per second to the *same* end-

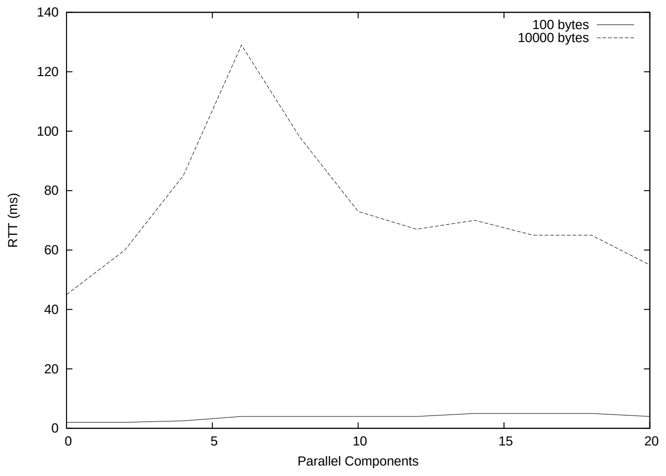

Figure 9: RTT vs load (other components)

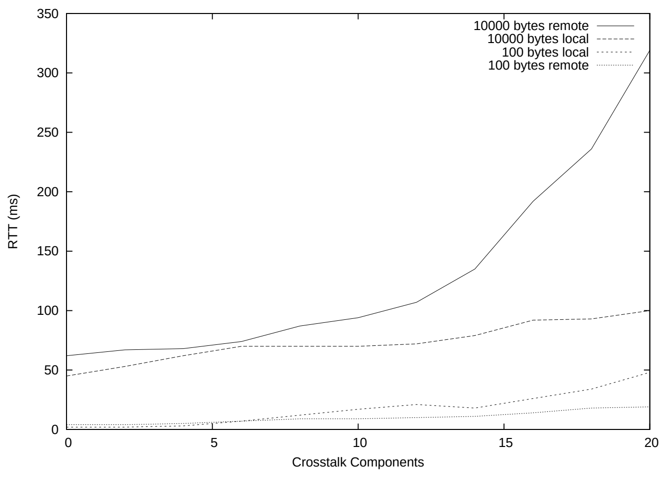

Figure 10: RTT vs load (same component)

point that is being used for the latency measurements. This tests the PIRATES wrapper's ability to keep up with incoming requests, rather than the OS kernel. Both local (same machine) and remote (LAN) scenarios are tested.

Due to the increasing queue lengths at the endpoint, response times increase approximately linearly with the degree of crosstalk. Interestingly the remote case starts with a RTT of 4ms (compared with 2ms for the local case) due to the extra network latency, but ultimately wins for small messages because the CPU load is then shared between two machines. For large message sizes the RTT increases superlinearly as the bandwidth starts to saturate the network link (10 Mbit/s).

11.4 Failure recovery

The RDCs maintain a set of *persistent components*, which must remain operational at all times. If one of these exits for any reason, an attempt is made to automatically restart it. Suitable replacements are located to satisfy the original component specification in the form of map constraints; thus the replacement may be on a different machine or a different program as long as it provides the same service. Once the replacement has registered, the mapping engine repairs its links to peer components.

The table in Figure 11 shows the time taken for persistent components to restart and begin communicating in different failure scenarios. The first column is the restart time for a single failed component, the second is the average restart time should ten components fail simultaneously.

Situation	One failure	Ten failures
Component deregisters	77 ms	330 ms
Connected component crashes	90 ms	700 ms
Isolated component crashes	5 secs	50 secs
Host machine reboots	2 mins	2 mins

Figure 11: Persistent component restart times

The quickest recovery occurs if the failed component[s] exit gracefully, deregistering themselves with the RDC. If they crash without deregistering and are not currently mapped to any other component, recovery takes at least five seconds because this is the default time period at which the RDC pings components to check they are still alive. Fortunately this case is rare in practice – components are not much use if they don't communicate, and hence are usually mapped to at least one peer. In this case distributed failure detection operates, and the peer component reports to the RDC immediately that it has lost contact. This allows the failed component to be restarted almost as quickly as if it had deregistered gracefully.

The most common multiple-failure case in which no mapped peers remain alive is when a machine hosting several components reboots; in this case all the components are restarted immediately once the host has recovered.

12. CONCLUSION

As part of the TIME project on transport monitoring, we have designed and implemented PIRATES, a middleware for processing streams of data from sensors within a conventional, powered network. The architecture uses wrappers which are both decentralised and decoupled, and hence combines the advantages of peer-to-peer and central event broker solutions. It is multi-modal, supporting events, RPCs and streams. Communication may be point-to-point or via intermediate brokers. We use an object data model with content-based subscriptions and fast type checking using a schema hash called a LITMUS code.

An advantage over traditional solutions is the comprehensive support for third-party remapping. We have found that this solves many common problems encountered whilst managing a continuously running and evolving large-scale sensor-based information system. Reflection and topology discovery are also provided by the wrappers. The implementation is portable and has a lightweight API; it can be downloaded from `http://www.srcf.ucam.org/~dmi1000/pirates`. Currently it runs on Linux and Mac OS.

Acknowledgements: The author would like to thank the other partners in the TIME project, and the EPSRC for their support.

13. REFERENCES

[1] Jean Bacon, Alastair Beresford, David Evans, David Ingram, Niki Trigoni, Alexandre Guitton, and Antonios Skordylis. Time: An open platform for capturing, processing and delivering transport-related data. In *Fifth IEEE Consumer Communications and Networking Conference, CCNC 2008, Session on Sensor Networks in Intelligent Transportation Systems*, Las Vegas, Nevada, US, January 2008.

[2] Guruduth Banavar, Marc Kaplan, Kelly Shaw, Robert Strom, Daniel Sturman, and Wei Tao. A case for message oriented middleware. In *Proceedings of DISC*, 1999.

[3] Guruduth Banavar, Marc Kaplan, Kelly Shaw, Robert Strom, Daniel Sturman, and Wei Tao. Information flow based event distribution middleware. In *Middleware Workshop at the International Conference on Distributed Computing Systems*, 1999.

[4] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Interfaces and algorithms for a wide-area event notification service. Technical Report CU-CS-888-99, Department of Computer Science, University of Colorado, May 2000.

[5] Paolo Costa, Geoff Coulson, Cecilia Mascolo, Gian Pietro Picco, and Stefanos Zachariadis. The RUNES middleware: A reconfigurable component-based approach to networked embedded systems. In *Proceedings of the 16th IEEE International Symposium on Personal Indoor and Mobile Radio Communications*, Berlin, September 2005.

[6] D-Bus home page. `http://dbus.freedesktop.org/`.

[7] Geraldine Fitzpatrick, Simon Kaplan, Tim Mansfield, David Arnold, and Bill Segall. Supporting public availability and accessibility with Elvin: Experiences and reflections. In *Computer Supported Cooperative Work*, volume 11. Springer, 2002.

[8] Phillip B. Gibbons, Brad Karp, Yan Ke, Suman Nath, and Srinivasan Seshan. IrisNet: An architecture for a world-wide sensor web. In *Pervasive Computing*, volume 2. IEEE, October 2003.

[9] C. Greenhalgh, S. Izadi, J. Mathrick, J. Humble, and I Taylor. ECT: A toolkit to support rapid construction of ubicomp environments. In *UbiSys*, 2004.

[10] Michi Henning. A new approach to object-oriented middleware. In *Internet Computing*. IEEE, January 2004.

[11] P. Hunt, D. Robertson, R. Bretherton, and R. Winton. SCOOT - a traffic responsive method of coordinating signals. Technical Report 1014, Transport Research Laboratory, UK, 1981.

[12] IBM WebSphere MQ home page. `http://www.ibm.com/software/integration/wmq/`.

[13] Internet Communications Engine home page. `http://www.zeroc.com/`.

[14] René Meier and Vinny Cahill. Taxonomy of distributed event-based programming systems. In *The Computer Journal*, pages 602–626. British Computer Society, June 2005.

[15] OASIS RELAX NG Technical Committeee. *RELAX NG Specification*, December 2001.

[16] OASIS Standard. *Universal Description, Discovery and Integration (UDDI) Version 3.0.2*, October 2004.

[17] OASIS Standard. *WS-BaseNotification 1.3*, October 2006.

[18] OASIS Standard. *WS-BrokeredNotification 1.3*, October 2006.

[19] Object Management Group. *Notification Service Specification, version 1.1*, October 2004.

[20] Peter R. Pietzuch and Jean M. Bacon. Hermes: A distributed event-based middleware architecture. In *Proc. of the 1st Int. Workshop on Distributed Event-Based Systems*, Vienna, Austria, July 2002.

[21] A. Rowstron, A-M. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *3rd International Workshop on Networked Group Communication*, UCL, London, Novermber 2001.

[22] SCOP home page. `http://www.srcf.ucam.org/~dmi1000/scop`.

[23] Bill Segall and David Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings AUUG97*, Brisbane, Australia, September 1997.

[24] Bill Segall, David Arnold, Julian Boot, Michael Henderson, and Ted Phelps. Content based routing with Elvin4. In *Proceedings AUUG2k*, Canberra, Australia, June 2000.

[25] Sun Microsystems. *Java Message Service Specification 1.1*, April 2002.

[26] W3C Recommendation. *XML Schema*, October 2004.

[27] W3C Recommendation. *SOAP version 1.2*, April 2007.

[28] W3C Recommendation. *Web Services Description Language (WSDL) V2.0*, June 2007.

[29] Daniel Wagner and Dieter Schmalstieg. Muddleware for prototyping mixed reality multiuser games. In *IEEE Virtual Reality Conference*, Charlotte NC, USA, March 2007.

[30] xmlBlaster home page. `http://www.xmlblaster.org/`.

[31] Yuanyuan Zhao and Rob Strom. Exploiting event stream interpretation in publish-subscribe systems. In *Principles of Distributed Computing*, 2001.