A pragmatic approach to middleware for distributed intelligent systems

Vincent Baines, Jeehang Lee, and Julian Padget

Department of Computer Science, University of Bath,
Bath, BA2 7AY, United Kingdom
{v.f.baines,j.lee,j.a.padget}@bath.ac.uk

Abstract Our objective is to support the rapid prototyping and deployment of distributed intelligent systems (DIS) in a research setting, where the components may often be legacy software not developed for a distributed environment. DIS are challenging to develop and test because they combine dynamic program behaviour with a dynamic environment and because the observations that can identify anomalous behaviour are often buried in large volumes of data. In order to mitigate these factors, we have constructed a simple loosely-coupled execution environment based on publish/subscribe communications that, to maximise return on effort both now and in the longer term, is realized by replaceable off-the-shelf components. By using a standardised protocol and standardised representations (JSON and RDF both over XML), combined with semantic annotation (RDF and RDF over JSON), components are further decoupled, in that an event stream need only contains tuples with the expected semantic tag to satisfy a given consumer process. Consequently, rather like Unix shell programs, event analysers, aggregators and visualizers can be incorporated into arbitrarily connected subscription networks. We present an evaluation of the framework[1] quantitatively through some performance data – our objective here is adequacy rather than high performance – and qualitatively through some case studies to assess accessibility, re-use and refactoring.

Keywords: Application of event-based middleware, Diagnosis and debugging of distributed applications, Distributed intelligent systems

1 Introduction

Since the development of Remote Procedure Call and later Remote Method Invocation, and yet later still with (SOAP-style) web services, much effort has been put into creating the illusion of being able to call a procedure on one machine, even if it resides on another. At the same time, various middlewares, including CORBA's event notification service, different enterprise service bus architectures (WebSphere, JBoss, Apache ServiceMix, Apache Camel, etc.) have offered forms of event notification. As Fielding [16] has argued and the designers of the above systems have implicitly contended, there is

[1] The framework components and example applications are available via `https://code.google.com/p/bsf/`, retrieved 20130524.

a poor fit between a synchronizing control mechanism and a high latency substrate, which is essentially the foundation of the case in favour of publish/subscribe (push/pull in CORBA) protocols on wide area networks.

The system we describe is a consequence of expediency as much as design: with limited (human) resources and wanting to put the majority of effort into building systems, rather than infrastructure, we have made a number of pragmatic choices to reduce risk, dependency on individuals and the volume of code we have to maintain, while increasing our reliance on the wider community. The central notion is to regard every component as a sensor – in line with the conventional description of a software agent as a component situated in an environment, from which it senses and on which it acts – whether the component is connected to the physical world or not.

Our objective is to support the capture and transport of data from a range of heterogenous devices and software components, where-ever they might be physically located, so that we can utilise various artificial intelligence techniques – specifically intelligent agents – across projects that involve: (i) interfacing to low level ad-hoc sensor networks, (ii) collecting, processing and rendering of energy consumption data[2], (iii) interfacing to and controlling avatars in virtual worlds [26], and (iv) interfacing to and controlling robots. Despite the difference in nature, these can all be viewed as event-based systems that consume and produce event streams.

In summary, the system requirements and their resolution can be stated as:

1. *Event-based notification over wide-area networks*: for which we use XMPP; this currently enjoys substantial support, with several high quality server implementations (ejabberd [13], prosody[30] and OpenFire[22], for example) and utilisation in internet messaging and other applications ([32,33], for example). We also note a study by Linden Labs[3], which shortlists XMPP as a suitable messaging technology, but which does not make a final recommendation.
2. *Standardized data representations*: for which we use both RDF and JSON; RDF is the only choice for the transport of semantically-annotated data, but does come with significant overheads, particularly in deserialization in which, rather like the Java class loader, it may be necessary to instantiate sections of a class hierarchy to support a particular triple; for this reason, and because of its widespread use as a serialized format – meaning there are many languages that offer (de)serialization libraries – we also use JSON.
3. *Capacity to support different programming languages and legacy applications*: interfaces employed so far support Java, C# and Python, but XMPP APIs exist for several other languages. We have also developed a means to interface to a RESTful service deployment mechanism [12], suitable for the combination of any command-line application and an appropriately configured virtual machine, but we do not discuss that further in this paper.

There is no novelty in the principles underlying the framework: depending on viewpoint, this could be seen as another take on the subsumption architecture [7] and, as we

[2] ENLITEN project: `http://gow.epsrc.ac.uk/NGBOViewGrant.aspx?GrantRef=EP/K002724/1`

[3] `http://wiki.secondlife.com/wiki/Message_Queue_Evaluation_Notes`, retrieved 20120416, last updated 2010.

have pointed out, we are using OTS (off-the-shelf) components, as much as we can, to meet our needs and limit wheel re-invention. There may be some novelty in the qualities deriving from this particular combination of facilities, particularly in respect of the support for semantic data streams, however, the main concern – and objection – we have encountered is whether adequate performance is possible along with a degree of scepticism about the practicality of the approach. Thus, the main contributions of this paper are: (i) how the combination of components addresses the requirements (ii) where performance issues may cause problems – and what can be done about them (iii) usability in respect of how additional components can be integrated to meet developers' needs as their requirements evolve, and (iv) usefulness, in which we review the adoption of the framework inside and outside our research group.

The remainder of the paper is organized as follows: we next (§2) review related work, then in §3 we describe the Bath Sensor Framework. The main contributions of the paper follow in §4, where we examine performance and (distributed) debugging issues and §5, where we review several cases of the application of the BSF to support the assessment of usability and usefulness. §6 concludes and outlines plans for current and future work.

2 Related Works

The demand for communication between one component and another has led to the development of a vast array of middleware architectures. Within this field, their use in connecting intelligent agents to simulated or real devices still includes a substantial number of approaches. The High Level Architecture (HLA) has been demonstrated in a wide variety of such applications in computer based simulations, commonly used in military training simulations [11] but also in multi-agent simulations using commercial games engines [29]. HLA has been established as a robust framework, however issues exist over the provision of the 'Run Time Infrastructure' (RTI) component where open-source implementations have historically lagged behind licensed alternatives. Whilst the equally established IEEE-specified Distributed Interactive Simulation (DIS) protocols exist for coupling simulation systems, HLA provides additional levels of simulation management with a key extension being time management [18].

Other middleware frameworks demonstrated in this domain bear comparison with the Bath Sensor Framework (BSF) middleware presented here. For example, the XCF framework [17] approach of XML messaging for distributed systems, and the CAST [20] application bringing disparate robotic system components together. Both of these are based on the ICE middleware framework [23] – in itself a strong middleware component – providing more advanced features that DCOM and CORBA solutions.

A contrasting approach to the use XMPP and its like, comes from the use of an Enterprise Service Bus. One example of the use of an ESB with agents [10] demonstrates the feasibility of the approach. We have no direct experience, but apparently Camel can be federated through combination with Apache ServiceMix.

With BSF middleware, there is no functionality in the sense of RPC/RMI type distribution, instead the focus is on the exchange of messages via the publish subscribe mechanism. This keeps the implementation lightweight, compared to the complexity

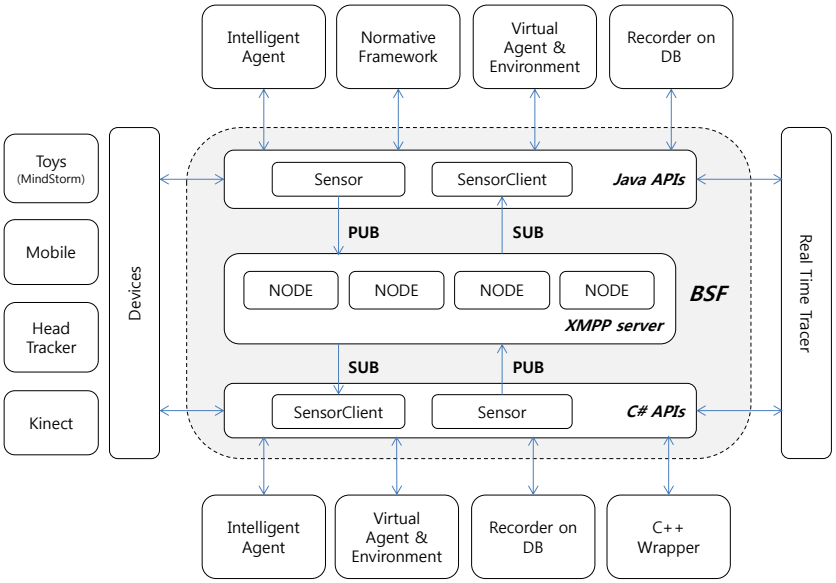

Figure 1. Bath Sensor Framework [24]

of the alternatives, e.g. defining HLA federations, managing ICE slice definitions. The focus is instead on the data itself, which is defined either in terms of RDF or JSON.

The matter of data format – on which XMPP itself is agnostic – introduces the topic of information exchange, which is explored further in section 5. In this theme, interest is in what information needs to be exchanged in order for an agent collective to succeed in a given goal. In the specific topic presented in this paper, the scenario is that of a vehicle convoy navigating to a specific destination. Such information is considered based on the work of Endsley [14], where the concept of exchanging 'low level' information such as position updates (e.g. perception level information) is compared to 'higher level' information such as what route is to be followed (e.g. projection level information), and impact of varying such communication strategies is presented in Section 5.

3 Middleware : Bath Sensor Framework

As mentioned earlier, Bath Sensor Framework (BSF) is based on the XMPP [34] communication layer. Although originally used in a somewhat different application, XMPP approaches have demonstrated in distributed computation applications [32,33], which is where the BSF model makes use of this approach. The open source openfire [22] messaging server is used to provide the publish-subscribe capability, with nodes used as communication channels between sensors publishing data, and subscribers to this data. A generic model of this is shown in Figure 1, in the context of a Jason BDI agent connecting to the BSF in order to receive sensor data from some remote entity and to send action commands to that entity based on its reasoning.

This architecture has been deployed in a number of configurations during the development of the applications put forward in Section 5. In its simplest form, as the only requirement is for an XMPP server and connected components, the BSF can be used on a standard laptop. However, as the complexity of the components grows, so demand for hardware resources increases, and it becomes prudent to distribute components across the network. Such expansion has been assessed on cloud computing resources as well, with the provision of a BSF service hosted on a cloud tier, with publishers and subscribers connecting to this service. Again, the issue of hardware occurs but this time in the sense of network latency. Compared to a local network, as cloud based servers require communication over the network, ping times were observed to shift by a factor of 20 (approximately 5ms to 100ms). The impact of this will vary with the nature of the application, but needs to be brought into consideration for the overall system architecture.

Further to the XMPP server, another component which has become integral to the BSF capability is an RDF logging service, currently provided by the Allegrograph RDF database software. An agent connects to the BSF and subscribes to any nodes of interest, and for every received RDF logs this into the database. This database store can then be queried directly (e.g. via SPARQL), or alternatively a number of components have been built to consume this data for other purposes. Within the 'rdfUtils' suite developed as part of the BSF, there is a replayer agent which retrieves RDF data from the database and broadcasts it back out to the BSF network. An analyser tool has also been developed, comprising of SPARQL queries along with programming logic, allowing powerful post simulation analysis. For example, in the context of the vehicle convoy scenarios in section 5, this allows analysis such as 'for each time step, for each convoy member, how close was the vehicle behind', where questions such as this can be adjusted and re-run without requiring the whole experiment to be repeated (with the results of such analysis shown later in Figures 8 and 9).

In the next section, we present the results of assessing both the performance of this framework and its debugging – because this also has lessons to offer, in part through the tools that were developed.

4 Performance and Debugging

Section 5 presents a number of applications which have been developed based on the BSF middleware. During the development of these, the behaviour of the underlying components of the BSF has become better understood through the process of debugging and capturing quantitative data on performance. The applications based on this middleware consist of a number of components in quite different formats i.e. from real world devices with high frequency low level sensors, to deliberative planning agents better suited to low frequency, high level communications. The issue of volume versus content links with the ideas presented in the previous section regarding a model that relates perception, comprehension and projection [14]: with high-frequency low-level information regarded as the perception level, and low-frequency high-level information regarded as the comprehension and projection stages.

We have found it convenient to consider the issues of performance and debugging using a layered approach, and although the labels we have used derive to our particular problem space, we believe the structure is applicable in any domain. Specifically, the layers are: (i) network: the lowest layer, (ii) (intelligent) agent: meaning the layer comprising the active components of the system, (iii) normative: meaning a representation of the rules that govern the behaviour of system components [9] (this may not be present explicitly in many systems, because such rules are often hard-coded into the components, but we regard their separation as key element in making system behaviour more flexible), and (finally) (iv) human: being the layer at which system actions are observed from the physical world. This structuring allows the breaking down of the system into smaller components, similar in approach to the standard testing approach of unit testing prior to integration testing. We now look at each of these layers in more detail as they pertain to the analysis and understanding of system behaviour.

4.1 Network Layer

As with any distributed system, establishing the network performance characteristics can be critical in understanding overall system behaviour. In publishing or subscribing to data streams, components may have requirements with respect to latency, message order consistency, and other aspects specific to the network layer and its performance. To avoid interface bloat, there is no in-built metric collection in the BSF implementation, but the data in each read operation allows for some integrity checking. Specifically, for each read, a timestamp is taken which is passed through to the subscriber with the data, permitting a check on message order, as no received reading should have a timestamp older than the previous reading. Checks can also be performed on the time taken for the message to be received, on which the subscriber can act (e.g. if this is deemed stale then perhaps discard it). However, feature creep can easily afflict such developments as such checks begin to move more into measurement of the overall network performance. To limit this effect, we have developed a standalone set of tools to capture data at this level, taking advantage of the fact that in notification frameworks it is a natural approach to outsource such functions to another subscriber component rather than fatten each interface.

One such tool is the ‘MonitorRDF’ package, which is shown in Figure 2. This presents a set of realtime graphs which contain measures of various BSF aspects, giving a human observer insight into the current behaviour of the system from the network layer perspective. This tool is also being developed as the basis for self-adaptive behaviour in message transmission rates, where the publisher can be asked to dynamically change its publishing rate.

MonitorRDF can be used to provide a realtime overview of the current simulation, but it can also save the current graphs to image files and csv data. In the experimentation performed to identify the maximum message throughput of a BSF experimental setup, this function was used to produce the results shown in Figure 3 and Figure 4. In these two graphs, a BSF test message (in this case a spatial position update) is sent at a specified frequency from the ‘rdfTest’ application. This application has been developed as a BSF ‘ping’ equivalent, where one instance is started as a publisher of BSF readings, and another instance is started as a subscriber. The subscriber instance outputs to the

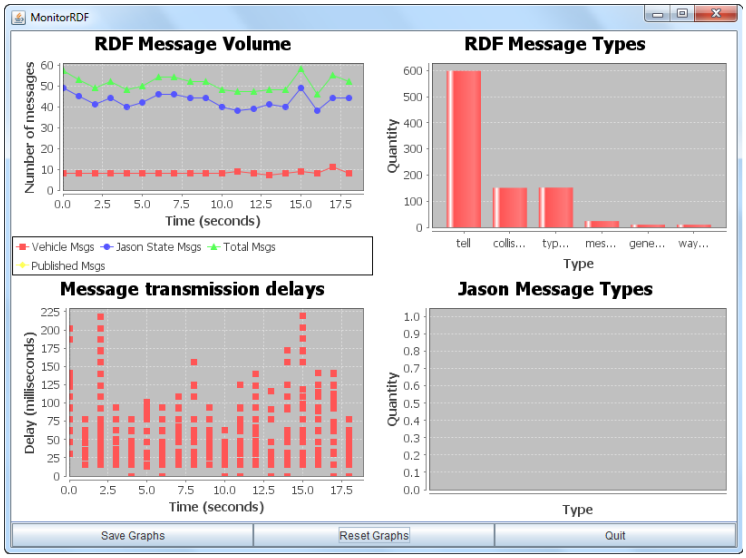

Figure 2. The MonitorRDF application

command line what delay has been incurred, and how many messages were received in the last time interval. This allows a baseline message throughput to be established, and Figure 3 and 4 reveal some issues: for the first 10 seconds, messages are published at 10/second, and in this period the expected number of messages were received, with low (below 100 millisecond) delays. At approximately 25 seconds, rdfTest started again, but publishing at 500/second. Figure 3 shows an increasing message delay, i.e. there is a backlog, while Figure 4 confirms this, with a sporadic quantity of messages processed each second.

The actual performance envelope obviously depends on the hardware involved. This tool helped us understand why one server was demonstrating poor message exchange rates (due to a particular wifi card). Perhaps unsurprisingly, wireless connections demonstrate significant variance in performance, as the results in Figure 4 show. For sake of space we omit the desktop graphs, but observe that the laptop subscription process peaks at a message rate of 170/second while the desktop hardware is stable at a message rate of 250/second. Publishing performance exhibits less sensitivity, perhaps as less logic is involved in the publish code, not that hardware does not have an impact: the peak publishing rate from the laptop is $\approx$600–700 messages/second, whereas the desktop peak publish rate is up to 6000 messages/second.

At the other end of the scale, a raspberry pi achieves a maximum message rate of between 70-80 messages/second, highlighting both the impact that a device's hardware specification can have, and also the need to consider that in a BSF application a component's mere participation does not necessarily ensure any level of message exchange performance and may affect overall performance in relatively unpredictable ways.

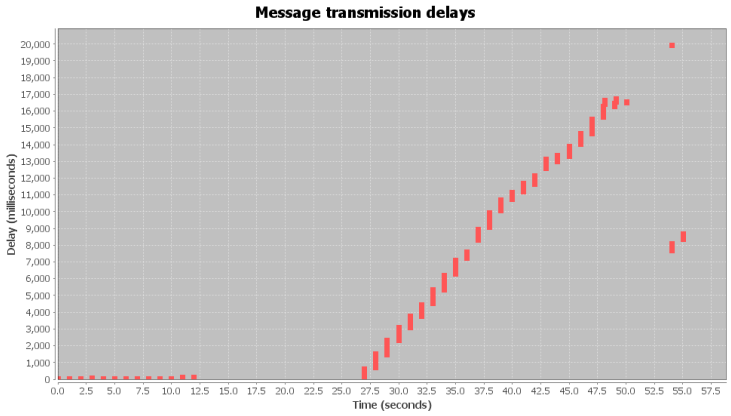

Figure 3. Message Delays

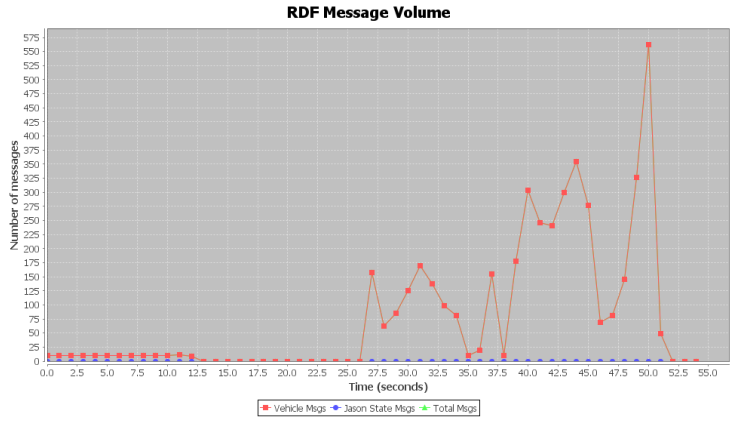

Figure 4. Message Volume

4.2 Agent Layer

In the applications in section 5, intelligent agents are connected to the BSF and interact with remote entities in order to enact plans and direct the behaviours of those entities. At present, the Jason [5] BDI architecture is the provider of the 'agent layer'. One benefit we have found arising from BSF is that the RDF triples can readily be examined to help understand the interaction between agents and the entities they control. Furthermore, the Jason architecture has been extended such that details of the agents 'mental state' are publish to the BSF, allowing real-time analysis based on agents' belief and plan usage. Monitoring components, such as the MonitorRDF tool can then subscribe to this data. Such data can also be used to augment a 3D view as shown in Figure 5, where the image has been annotated with 3 numerical markers, to be interpreted as follows: (i) by marker '1', there is a square object, which has been dynamically created

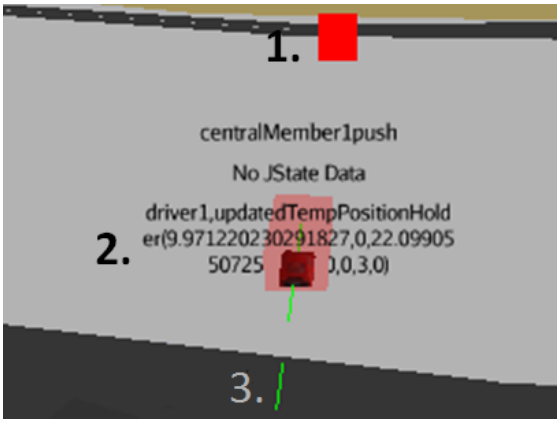

Figure 5. Jason agent mind state augmentation

when the agent added a new belief of 'destination(x,y,z)' (with the object displayed at correct x, y, z location in the 3D view) (ii) the block of text next to marker '2' shows the vehicle's agent's name, and any Jason messages it has currently received (in this case updatedTempPositionHolder values have changed); there is also a semi-transparent volume, which indicates the currently calculated collision volume, (iii) next to marker '3', the green line corresponds to the previous collision volumes (as the vehicle has been moving from the bottom of the image towards marker '1'.

This illustration is intentionally highly application specific because what we aim to demonstrate is the degree of flexibility provided by the framework in respect of the variety and purpose of message exchanges. Specific data channels can be established for publishing certain data types (e.g. vehicle positions), which is separate from other data (e.g. Jason mind state data), with interested parties subscribing to their channel(s) of interest.

4.3 Normative Layer

The normative layer follows the approaches set out in [1] and [3], which in turn builds on the formal and computing model described by [8], to represent the governance and institutional framework around intelligent agents situated in virtual environments. The agent layer is free to pursue given goals using whatever plans are available to it, which provides the intelligence and autonomy behind the decision making of virtual agents. However, there are situations where this plan selection may need to be constrained, in order to bring it in line with some social convention (e.g. be polite and let a vehicle join the convoy in front of you) and also to improve the group performance (e.g. in a traffic jam, all vehicles cooperate to attempt to alleviate the congestion).

Such a control mechanism adds a further challenge to understanding observed vehicle behaviour, as the reasoning about context is now distributed between the agent themselves (e.g. slowing down to prevent a collision) and a governance framework (e.g.

slowing down to improve convoy performance): again this is hard to link to observations and hence debug. Furthermore, the normative layer may subscribed to data streams that the agent layer is not processing, and use these data sources to enforce behaviours. As mentioned earlier in the middleware discussions, it may be inappropriate for the agent layer to handle large volumes of high frequency data, but this may be suitable for the normative layer. For example, stream reasoning [19,2,4] could allow this layer to subscribe to all vehicle positions and identify higher level issues to improve convoy performance (e.g. benefits in reaction to traffic patterns [15]). This information could then be used to enforce a response via the agent layer. Such an implemented scenario is described in Section 5.4

Action selection such as this, may then be displayed via the 3D viewer in order to provide the human observer with a visual cue to understand better what is occurring in the simulation, i.e. that vehicle behaviour may be different to expectation as some normative behaviour is being enforced.

4.4 Human Layer

The human layer is responsible for the task of representing aspects of the simulation state in visual form such that a human observer can infer some understanding of what is occurring. This can be in the form of a graphical representation of the data being transmitted over the middleware, e.g. rendering a 3D scene, or alternatively characteristics of the middleware itself can be displayed, e.g. message transmission delays in graph format such as that shown earlier in Figure 2.

This layer typically combines several data sources in order to assist the human observer in understanding the application state, for example Figure 6, where a 3D view has been developed for representing information from the vehicle convoy scenarios discussed here. In another illustration of the customisable nature, a bar chart in the top right corner displays the current number of BSF messages received per second. If BSF data is being replayed from a recorded simulation via the BSF replayer tool, then the buttons shown in the bottom of Figure 6 enable the operator to pause, rewind, and resume the visualisation.The combined effect of these capabilities is that the human operator is able to interpret the state of the simulation through selected observed events. This has demonstrable benefits in debugging, where issues can be caused by events in the agent layer (e.g. an incorrect belief, unsuitable plan selection), network layer (e.g. saturation of BSF messages, transmission latency) or normative aspects. Furthermore, the interaction of issues between layers may be observed via this tool, such as intervals in the XMPP graph where no messages have arrived explaining why a vehicle did not take some action.

5 Research applications

This section discusses two applications developed using the BSF middleware and sketches some related projects for the sake of the lessons learned. As well as demonstrating the framework in use, the motivation of developing such applications is to understand the

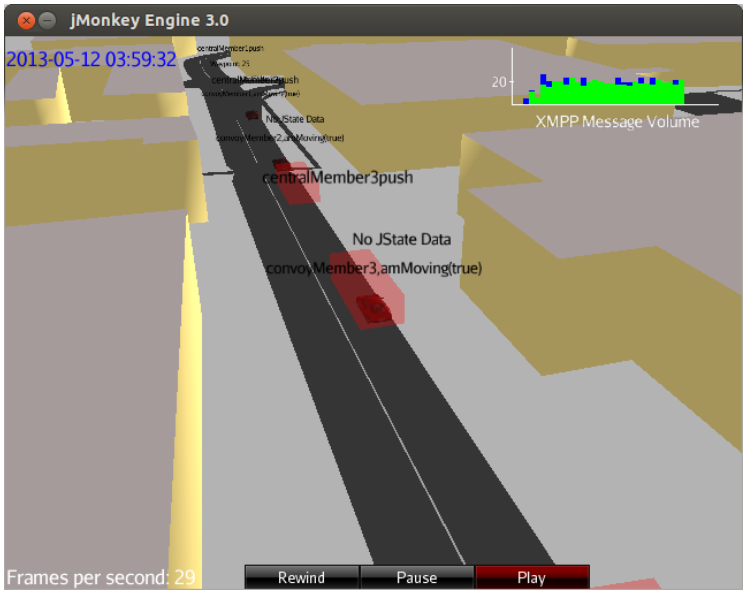

Figure 6. Bath Sensor Framework - 3D View Tool

performance characteristics and capabilities of this framework in real applications dealing with real world issues such as network latency, bandwidth limitations, and CPU bounding.

5.1 Intelligent convoys

The aim of this project is to explore the effectiveness of exchanging higher level knowledge between entities while communicating less and to investigate how this impacts collective understanding. The motivating scenario is convoys of intelligent vehicles.

System composition An overview of the system components showing their data subscription and publishing communication, along with indicative message volume and transfer rates, is given in Figure 7. As can be seen in this figure, the BSF middleware is at the heart of the composition, with a number of additional components specific to this scenario built around that framework. In this scenario, vehicles are represented by components that respond to commands received (e.g. setOrientation, setSpeed) and publish geospatial data to the framework to which other interested parties may subscribe. In this case, the key component subscribed to this geospatial information is the intelligence layer provided by Jason. On receipt of this data, the Jason BDI agents are updated with new beliefs, which activates plans thus generating vehicle commands that are published via the BSF.

Whilst this provides the core solution for coupling intelligent agents with remote vehicles, a number of additional components are used that bring significant benefit for

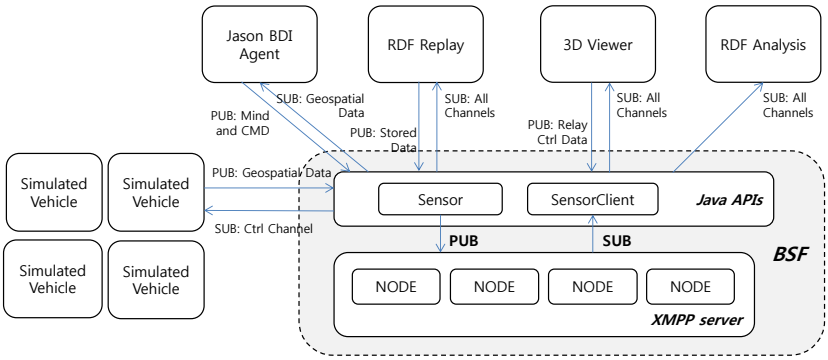

Figure 7. System Components of Intelligent Convoy

the developer. Firstly, there is a 3D viewer (based the jMonkeyEngine Java framework), the details of which have largely been covered in section 4.

Convoy scenarios We have implemented three convoy scenarios so far, the first two of which provide a data volume and performance baseline, while the third starts the exploration of 'less but better' communication. In the first scenario, position data is pushed from the vehicle ahead to the vehicle behind and the second vehicle takes that position as its next goal. Updates are published at fixed intervals and there is no (dynamic) control over transfer rates. The second scenario is the inverse in that the vehicle behind requests (pulls) the position of the vehicle in front and uses the response to set its next goal. This offers slightly more control – and reduced data volumes – because data is requested on need. However, in both scenarios, the data is the same, namely a position, which considered in terms of Endsley's [14] model (Section 4) is quite low level. Although pull puts less load on the middleware than push, communication overheads are relatively high because of the quality of the data communicated. The third scenario only communicates waypoint information. The lead vehicle 'knows' (i.e. has previously computed) the route and informs the rest of the convoy of the details (the waypoints to move to).

Scenario results For sake of space and because scenarios one and two are very similar, we only show results for the push (Figure 8) and waypoint (Figure 9) scenarios. The graphs show that similar convoy gaps are maintained between the four vehicles in both scenarios, but there are notable differences in acceleration and braking, with apparently much more micro adjustment taking place in the first scenario. This is due to scenario three vehicles following a route that it is a copy of the lead vehicle's route, while in scenario one the last known position leads to corner-cutting, greater inter-vehicle gap change and hence actions to re-establish the defined gap.

The entire experimental framework used in these experiments is available for download from the URL given on the first page. Furthermore, the specific data sets captured

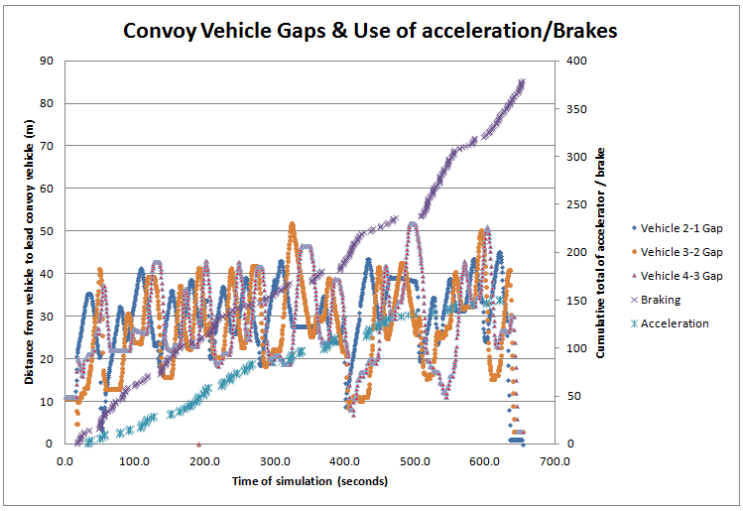

Figure 8. Convoy 'Push' Scenario

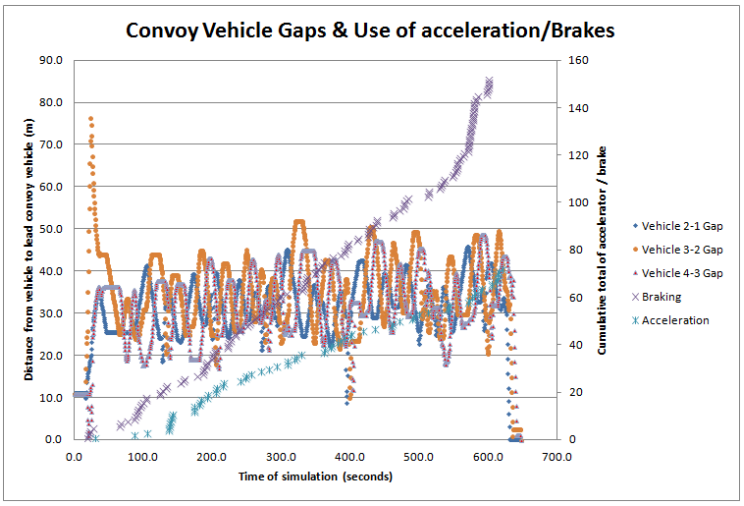

Figure 9. Convoy 'Waypoint' Scenario

and analysed in these three scenarios are available, and can be replayed via the BSF and the rdfReplayer component.

5.2 Intelligent Virtual Agents in *Second Life*

The aim of this work is imbue more believable behaviours into virtual characters with regards to improved awareness of social situations. The approach we have adopted uses

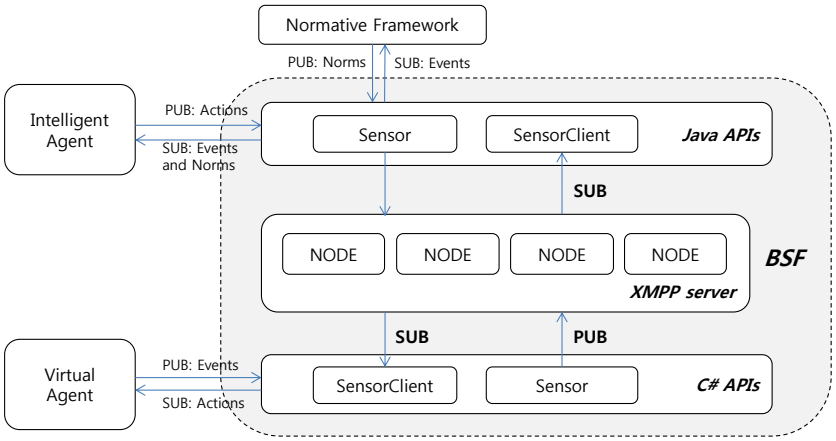

Figure 10. Overview of System Components of IVAs

a combination of individual reasoning, carried out by cognitive agents, and social reasoning carried out by a normative framework, as described in section 4.3. This objective is accomplished by the distributed agent platform [25], which is another example of a distributed intelligent system whose integration is achieved by means of BSF. The agent platform comprises the virtual agents (VA) in Second Life, an institutional model for social reasoning, and Jason BDI agents that are responsible for individual reasoning in response to normative positions published by the institution.

As with the intelligent convoy, the use of BSF is a pragmatic choice for the integration of a whole system due to the heterogeneity of the software components involved, as well as the use of different programming languages. In contrast to the convoy scenario, however, a higher level of knowledge is typically used in messages in order to reduce communication loads and because it better reflects the level at which the agents operate.

System Composition An overview of the distributed intelligent systems for intelligent virtual agents is illustrated in Figure 10. Given the characteristics of BSF, all data communications amongst them are performed using event based publish-subscribe mechanism.

The virtual agents (VA) are viewed as a sort of sensor for the entire system. Once the external events are perceived from *Second Life* (*SL*), the VA interprets these events into symbolic representations and publishes them, while both the BDI agent and the institution subscribe to that stream. When the institution received this information, it triggers the social reasoning process, which results in changes in the normative positions of the actors and which identifies appropriate behaviours for the current (social) situation. This information is then published as permissions or obligations for the BDI agent to incorporate into its reasoning process. When the decision making process completes, the action plans are published, which is realised with the sequence of atomic actions by the VA. For more details, please see [25].

(a) Queuing (b) Detecting (c) Yielding

Figure 11. Queuing [25]

Figure 12. Politeness [26]

Experiments For the experimental set-up, we use the OpenMetaverse library [28] to establish a control link with the virtual agents in the *Second Life* virtual environment, while the Jason platform, as before, is used for the intelligent agents. Additionally, the normative framework is supported by the institutional model proposed by Cliffe *el al.* [8]. All those entities are coupled via BSF on Openfire. Given these settings, we demonstrate two scenarios: queueing model introduced in [25], and modelling the politeness in IVAs in a dense crowd [26], shown in Figure 11 and 12, respectively. [4]

Apart from the quality of decision making and behaviours, we believe that the challenge here lies in the middleware perspective: the distibuted agent platform is composed of heterogeneous software components in terms of programming languages. The VA is implemented in C# and Jason and the institution software is in Java. In addition, some information (e.g. symbolic representations of external events) should be shared at the same time between multiple components, which should be coupled as if it were a single process application to achieve an acceptable response time, but loosely coupled for potential extensions by other entities.

In this circumstance, it is assumed that the use of BSF is affordable choice to resolve such challenges. Given multiple-language support, the heterogeneity issue is resolved by protocol-based integration, which is accomplished in practice by the pub/sub functionality of the C# and java XMPP libraries, MatriX [27] and Smack [21], respec-

[4] The video clips of the queueing and politeness model are available via `http://www.cs.bath.ac.uk/~jl495/queue.wmv` and `http://www.cs.bath.ac.uk/~jl495/politeness.wmv`, respectively, retrieved 20130530.

	Elapsed Time			
	Published every $25ms$		Published every $100ms$	
	μ	σ	μ	σ
Java to Java	1.875 ms	0.625	-	-
C# to Java	1.119 ms	0.630	-	-
Java to C#	2.826 ms	7.557	1.156 ms	2.268
C# to C#	2.664 ms	7.829	1.150 ms	2.466

Table 1. Delivery Time

tively. Starting from the concept of the sensor-based programming model illustrated in [24], the agent platform is established simply by adding *Sensor* and *SensorClient* objects, whose task is to publish and to subscribe, respectively, to information from a specified node in the XMPP server. Regarding the representation of data, this scenario uses JSON instead of RDF due to its lightweight characteristics in terms of serialisation/deserialisation.

Evaluation We have carried out some basic performance evaluation by measuring the elapsed time during the publication and subscription, specifically checking the time just before the publication of data that is already packed as a single item, and taking the time when the subscription handler detects the arrival of the item. Two cases are explored between (i) java entities, e.g. BDI agent and the institution, and (ii) java and C# entities, e.g. the VA and BDI agent or vice versa, and the VA and the institution,

For the sake of the integrity of the evaluation, we use exactly the same data as generated by the convoy model, [5]. During each evaluation, 152139 items (2 elements, metrics and commands each) are published every 25msec. No data losses are observed on the subscriber side. The overall statistics are shown in Table 1, where μ and σ are the mean and standard deviation, respectively. These results suggest that transport latency in the case of this data set is very low. Based on the largest average elapsed time (2.826ms), this implies a frame update rate of at least 350 fps, not accounting for time spent on either plan processing or scene rendering. In practice, we observe that subscription in Java applications seems both faster and more reliable, in the sense that the standard deviation is much lower that for C#. C# also appears to be more sensitive to the publication rate, since if the interval is increased (from 25ms to 100ms), the average time falls slightly and the standard deviation improves significantly.

5.3 Collected small projects

Consumer energy displays the framework has been used in two six week experiments (on the University of Bath campus) on occupant response to the presentation of energy consumption information. In this case the 'sensors' are the meter feeds stored in a Microsoft SQL server, extracted using a script in sqsh (SQL shell on Linux), published

[5] The full test set is available via `https://code.google.com/p/bsf/downloads/list`, retrieved 20130530

to `jabber.org` and subscribed to by an Android app running on tablets mounted in student kitchens. Subsequently, the same set-up was deployed for several months in a commercial environment. Results on response to display type and impact on energy behaviour are now under review.

Microsoft Kinect is very popular as an interaction device, since it is able to detect several kinds of human behaviours (e.g. free hand gestures, speech recognition etc). This is useful to distributed intelligent systems as a means of human - agent interaction. In this case, the system is constructed with the simple addition of *Sensor* object in the Kinect application. Once free hand gestures are detected by the Kinect, they are turned into a symbolic representation, which is published via the *Sensor* object to the (human controlled) avatar. This avatar then triggers corresponding actions of the virtual humans governed by the distributed agent platform presented in Section 5.2[6].

Robots of various kinds A Parrot arDrone Quadracopter, which is normally controlled by a smartphone app over wifi, to which it relays a video feed from a forward-facing camera, and from which the user controls movement. In this case the smart phone is replaced by a BSF bridge component, which publishes the data from its geo-spatial sensors and subscribes to control commands from the BSF, relaying them to the arDrone. Additionally it converts the arDrone video feed to an RTMP video stream and sends this to the OpenFire server, thus allowing BSF components to connect to this URL and access the video stream for display, analysis, as well as control via novel means (with work ongoing to use an Xbox Kinect interface) as above. The LEGO Mindstorms robot has a bluetooth interface, so by combining it with an Android phone, which runs a BSF interface, it becomes possible to build physical autonomous convoys [6], where the primary reasoning can be carried out by agents on a desktop computer that publish actions for the Android handset to interpret and break down into commands suitable for the Mindstorms controller.

5.4 Retro-fitting

The experience reports so far have described developments that were constructed with the use of the BSF as part of the plan. The final contribution in this section examines a case where the framework has been used in a major refactoring and extension of an existing distributed (AI) application. In [31], Ranathunga et al describe an event based architecture for the control by an intelligent agent of an avatar in the Second Life soccer scenario. This was realized by several software components: (i) a snapshot generator that coalesces low-level sensor data from SL, (ii) the relation identifier that adds associations between events, (iii) the complex event detector (written in Esper[7]), as well as (iv) the SL interface and (v) the (controlling) agent platform. All of these

[6] The video clip is available via `http://cs.bath.ac.uk/~jl495/kinect.wmv`, retrieved 20130530

[7] event stream processing engine from `http://esper.codehaus.org/`, retrieved 20130531

processes were set up on a single machine using a manually configured set of TCP/IP connections. Performance was never an issue, but the system was fragile and adding new components a delicate matter, where a failure in one part could lead to system lock-up and an extensive debug cycle. A further issue was the effort involved in interfacing components written in different languages. Consequently, the system was refactored to utilise the BSF over a period of approximately a month and subsequently augmented by the integration of two new components: (i) the natural language processor that analyses chat channel data, and (ii) the situation identifier.

It bears repetition that one consequence of the kind of framework now employed is that consumer and producer do not have to be aware of each others' identities, whereas the socket-based system necessarily encodes knowledge of one process in another. Thus, there are three usability observations from this story: (i) the relative ease with with an existing distributed application could utilise the new framework, (ii) the development of a pub/sub API for a given language simplifies the integration of every component in that language, and (iii) that looser coupling has improved the capacity for integration.

6 Conclusion and future work

Our objective has been to re-use existing wheels as effectively as possible and not to re-invent them, thus we have tried very much to apply existing event-based middleware in the context of a range of research around the topic of distributed intelligent systems that are, on the one hand quite diverse in terms of the problems they address, but on the other have sufficient commonalities that a substantial class of problems can be captured in a common framework. By viewing the middleware as a commodity, we have been able to concentrate effort on building the interfaces and tools for diagnosing the behaviour of small scale distributed applications, and as our experiences have shown, address some of the very specific requirements that debugging takes on when operating in this kind of context, where often building the right visualization is the only way to make sense of the complex states created by a collection of non-deterministic components. The notion of layers, inspired by networking, has particularly helped as a kind of litmus test to decide where different functionality belongs and how to decide what information to reveal where and what to encapsulate. Finally, we are particularly encouraged that not only does the framework help in developing new applications, it can also be used retrospectively to improve a 'legacy' style distributed application based on direct socket communications.

Future work includes the development of generic monitoring tools – as we realize from practice what these are – for analysing data flows. Subsequently data collected about the system by itself can be delivered to itself as sensor feeds so that a cadre of system management agents might be able to take decisions about the running, suspension and migration of parts of the platform, leading to dynamic configuration and self-management subject to normative governance. Furthermore, since that governance is represented as data, it could be changed over time, either externally by administrators, but more interesting is the potential for group decision-making between the system agents to determine policy changes themselves.

References

1. Natasha Alechina, Mehdi Dastani, and Brian Logan. Programming norm-aware agents. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, AAMAS '12, pages 1057–1064, Richland, SC, 2012. International Foundation for Autonomous Agents and Multiagent Systems.
2. Darko Anicic, Paul Fodor, Sebastian Rudolph, Roland Stühmer, Nenad Stojanovic, and Rudi Studer. A rule-based language for complex event processing and reasoning. In Pascal Hitzler and Thomas Lukasiewicz, editors, *RR*, volume 6333 of *Lecture Notes in Computer Science*, pages 42–57. Springer, 2010.
3. T. Balke, M. De Vos, J. Padget, and D. Traskas. On-line reasoning for institutionally-situated bdi agents. In *The 10th International Conference on Autonomous Agents and Multiagent Systems - Volume 3*, AAMAS '11, pages 1109–1110, Richland, SC, 2011. International Foundation for Autonomous Agents and Multiagent Systems.
4. D. Barbieri, D. Braga, S. Ceri, E.D. Valle, Yi Huang, V. Tresp, A. Rettinger, and H. Wermser. Deductive and inductive stream reasoning for semantic social media analytics. *Intelligent Systems, IEEE*, 25(6):32 –41, nov.-dec. 2010.
5. R.H. Bordini, M. Wooldridge, and J.F. Hübner. *Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons, 2007.
6. Benjamin Bourdin. Distributed agent architecture for autonomous convoys. Master's thesis, University of Bath, UK, 2012.
7. R.A. Brooks. A robust layered control system for a mobile robot. *Robotics and Automation, IEEE Journal of*, 2(1):14–23, 1986.
8. O. Cliffe, M. De Vos, and J. Padget. Specifying and reasoning about multiple institutions. In Pablo Noriega, Javier Vázquez-Salceda, Guido Boella, Olivier Boissier, Virginia Dignum, Nicoletta Fornara, and Eric Matson, editors, *Coordination, Organizations, Institutions, and Norms in Agent Systems II*, volume 4386 of *Lecture Notes in Computer Science*, pages 67–85. Springer Berlin Heidelberg, 2007.
9. Owen Cliffe, Marina De Vos, and Julian Padget. Modelling normative frameworks using answer set programing. In Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors, *LPNMR*, volume 5753 of *Lecture Notes in Computer Science*, pages 548–553. Springer, 2009.
10. Stephen Cranefield and Surangika Ranathunga. Embedding agents in business applications using enterprise integration patterns. In *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*, AAMAS '13, pages 1223–1224, Richland, SC, 2013. International Foundation for Autonomous Agents and Multiagent Systems.
11. Adam Szydłowski Dariusz Pierzchała, Michał Dyk. Distributed military simulation augmented by computational collective intelligence. In Kiem Hoang Piotr Jędrzejowicz, Ngoc Thanh Nguyen, editor, *Computational Collective Intelligence. Technologies and Applications*, volume 6922 of *Lecture Notes in Computer Science*, pages 399–408. Springer Berlin Heidelberg, 2011.
12. Kewei Duan, Julian Padget, H. Alicia Kim, and Hiroshi Hosobe. Composition of engineering web services with universal distributed data-flows framework based on ROA. In *Proceedings of the Third International Workshop on RESTful Design*, WS-REST '12, pages 41–48, New York, NY, USA, 2012. ACM.
13. ejabberd community site. ejabberd community site homepage. Retrieved from `http://www.ejabberd.im/`,, 20130507.
14. Mica R. Endsley. Toward a theory of situation awareness in dynamic systems. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 37(1):32–64, 1995.
15. Sándor P. Fekete, Christiane Schmidt, Axel Wegener, Horst Hellbrück, and Stefan Fischer. Empowered by wireless communication: Distributed methods for self-organizing traffic collectives. *ACM Trans. Auton. Adapt. Syst.*, 5(3):11:1–11:30, Sep 2010.

16. Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.
17. Jannik Fritsch and Sebastian Wrede. An integration framework for developing interactive robots. In *Software Engineering for Experimental Robotics*, pages 291–305. Springer, 2007.
18. Richard M Fujimoto. Time management in the high level architecture. *Simulation*, 71(6):388–400, 1998.
19. Martin Gebser, Torsten Grote, Roland Kaminski, Philipp Obermeier, Orkunt Sabuncu, and Torsten Schaub. Stream reasoning with answer set programming: Preliminary report. In Gerhard Brewka, Thomas Eiter, and Sheila A. McIlraith, editors, *KR*. AAAI Press, 2012.
20. Nick Hawes and Marc Hanheide. CAST: Middleware for memory-based architectures. In *Proceedings of the AAAI Robotics Workshop: Enabling Intelligence Through Middlware*, July 2010.
21. Ignite Realtime. Ignite realtime smack api homepage. Retrieved from `http://www.igniterealtime.org/projects/smack/`, 20130129.
22. Ignite Realtime. The Openfire Project. `http://www.igniterealtime.org/projects/openfire/`, 20130129.
23. ZeroC Inc. The internet communications engine, 2005.
24. Jeehang Lee, Vincent Baines, and Julian Padget. Decoupling cognitive agents and virtual environments. In Frank Dignum, Cyril Brom, Koen Hindriks, Martin Beer, and Deborah Richards, editors, *Cognitive Agents for Virtual Environments*, volume 7764 of *Lecture Notes in Computer Science*, pages 17–36. Springer Berlin Heidelberg, 2013.
25. Jeehang Lee, Tingting Li, Marina De Vos, and Julian Padget. Governing intelligent virtual agent behaviour with norms. In *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*, AAMAS '13, pages 1205–1206, Richland, SC, 2013. International Foundation for Autonomous Agents and Multiagent Systems.
26. JeeHang Lee, Tingting Li, and Julian Padget. Towards polite virtual agents using social reasoning techniques. *Computer Animation and Virtual Worlds*, 24(3-4):335–343, 2013.
27. Matrix XMPP Library. Matrix xmpp sdk homepage. Retrieved from `http://www.ag-software.net/matrix-xmpp-sdk/`, 20130507, no date.
28. OpenMetaverse Organization. libopenmetaverse developer wiki homepage. `http://lib.openmetaverse.org/wiki/`. Retrieved 20121217.
29. Tomás Plch, Tomas Jedlicka, and Cyril Brom. Hla proxy: Towards connecting agents to virtual environments by means of high level architecture (hla). In Frank Dignum, Cyril Brom, Koen V. Hindriks, Martin D. Beer, and Deborah Richards, editors, *CAVE*, volume 7764 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2012.
30. The Prosody Project. Prosody im. `http://prosody.im/`, 20130531. retrieved 20130531.
31. Surangika Ranathunga, Stephen Cranefield, and Martin Purvis. Identifying events taking place in Second Life virtual environments. *Applied Artificial Intelligence*, 26(1–2):137–181, 2012.
32. L. Stout, Michael A. Murphy, and S. Goasguen. Kestrel: an XMPP-based framework for many task computing applications. In *Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers*, MTAGS '09, pages 11:1–11:6, New York, NY, USA, 2009. ACM.
33. J. Wagener, O. Spjuth, E. Willighagen, and J. Wikberg. XMPP for cloud computing in bioinformatics supporting discovery and invocation of asynchronous web services. *BMC Bioinformatics*, 10(1):279, 2009.
34. XMPP Standards Foundation. The XMPP standards foundation homepage. Retrieved from `http://www.xmpp.org`,, 20130129.