An Open Meteorological Alerting System: Issues and Solutions

Ian Mathieson[a] **Sandy Dance**[b] **Lin Padgham**[a] **Malcolm Gorman**[b]
Michael Winikoff[a]

[a]School of Computer Science and Information Technology,
RMIT University,
GPO Box 2476V, Melbourne, VIC 3001
Email: {idm,linpa,winikoff}@cs.rmit.edu.au

[b]Australian Bureau of Meteorology,
Melbourne, VIC 3001
Email: {s.dance,m.gorman}@bom.gov.au

Abstract

This paper describes an experimental alerting system under development by the Australian Bureau of Meteorology, initially targetted at (but not restricted to) the aviation sector. The system provides alert routing and filtering: for example pressure readings from automated weather stations may conflict with a local terminal aerodrome forecast, resulting in an alert being displayed to forecasters and other interested parties (such as airlines or individual aircraft).

The multi-agent based design is inherently distributed and readily facilitates scalability and system evolution by simplifying integration of new services and components: for example, adding new types of data sources and/or alerts spanning multiple organisations and system platforms. Another key issue is robustness: the system must be able to adapt to failure of individual components.

Further issues that arise concern more user-focussed alert provision: an aircraft may wish to be notified about alerts (or new alert types) that concern it, i.e. that take place in certain regions.

In this paper we present the design of the system, discuss how the design addresses some of the issues, and outline our plans for supporting more flexible alert notification. Some early evaluation trials are currently underway.

Keywords: Artificial Intelligence, Distributed Systems, Real-time Systems, Software Engineering.

1 Introduction

The Australian Bureau of Meteorology[1] headquartered in Melbourne is the national weather service of Australia. It has a strong need for complex and evolving systems for managing its weather forecasting, monitoring and alerts and is currently in the process of developing a sophisticated software system in which intelligent agents play a significant role.

This work was funded by ARC Linkage grant LP0347925 (*Open Agent Architectures for Intelligent Distributed Decision Making*), a joint research grant involving the RMIT University Agents Group, the Australian Bureau of Meteorology and Agent Oriented Software.

[1]http://www.bom.gov.au

There are a number of challenges to be met during this development:

- The system must evolve over time. It must include legacy software, and must include and make use of new and more sophisticated components as these are made available.
- It must be a distributed and open system. Components must be able to run on different platforms, and must be able to be developed and deployed by different groups with only loose cooperation. As new components are added they must be located and used appropriately.
- The system must handle large amounts of data, used and produced by many components including legacy software.
- The system involves a range of complex goals, a highly dynamic environment and some complex inferencing.

The Forecast Streamlining and Enhancement Project (FSEP) is a major project within the Australian Bureau of Meteorology which seeks to improve the quality, quantity, consistency and timeliness of weather products and services to the community and major clients such as the aviation industry, fire fighters and emergency services. Additional potential clients include shipping and agriculture. Increasingly, clients require real-time alerting of significant weather events. To improve the timeliness of weather alerts to clients, and to help streamline the work-flow of forecasters, intelligent alerting within the forecast system has a high priority in FSEP.

The domain is highly dynamic, with large amounts of data about a (sometimes) rapidly changing environment. There are also a wide range of tasks that must be addressed by the system, such as detecting particular meteorological phenomena, resolving inconsistencies in information, providing appropriate focused information to users, closely watching particular geographical areas (e.g. around airports), etc. This combination of a range of complex tasks and a highly dynamic environment, makes a system incorporating intelligent agents, which can be both reactive and proactive, a natural choice.

In this article, we first introduce intelligent software agents, before exploring the application domain further. We then describe our initial implementation, built using a commercial agent framework, and show how it addresses a number of these requirements. Finally, we outline our requirement for flexible alert notification and indicate some promising research leads currently under investigation.

2 BDI agent systems

The research ideas behind *intelligent agents* date back to the mid 1980's (Georgeff & Lansky 1986, Bratman 1987). The technology has subsequently been used successfully in a number of challenging applications such as air traffic control (Ljungberg & Lucas 1992) and space shuttle monitoring (Georgeff & Ingrand 1990).

We are particularly interested in *goal directed agents* using pre-specified *plans*, such as those supported by the agent development frameworks JACK Intelligent Agents™ (Busetta et al. 1999), dMARS (d'Inverno et al. 1998), PRS (Ingrand, Georgeff & Rao 1992), JAM (Huber 1999), etc. These are referred to as *BDI* (Belief, Desire, Intention) agents, because of the way that they represent and work in terms of these kind of concepts. (In particular we use JACK, an industrial strength system for developing BDI agents, developed by Agent-Oriented Software[2] in Melbourne, Australia.)

The plans in these systems describe a particular way of achieving a goal (or sub-goal) in a particular situation, known as a *context*. The goal directed nature of the agent execution mechanism provided by the system ensures that if an agent fails to achieve its goal using a particular plan, it will search its plan library and try an alternative plan if one is available. Appropriate plans for use are decided as late as possible, i.e. only when the agent is ready to achieve the sub-goal. Thus the choice is always made with the current situation in mind. Plans can also contain sub-goals, which allows for a hierarchical approach where goals are broken down into sub-goals, which may themselves be further broken down. At each level the appropriate plan for the current situation is chosen from the library of available plans.

The combination of reactivity and goal-directedness of BDI agent systems makes them an excellent candidate for complex applications operating in dynamic environments. In addition to this run-time flexibility, BDI systems are highly scalable. As new situations are identified and ways of behaving in those situations developed, additional plans can be added to an agent's repertoire, along with a description of the applicable situation.

2.1 An agent development toolkit

JACK is a third generation BDI agent system built on top of Java and includes:

- An agent-oriented programming language that extends Java with agent concepts
- Infrastructure for running distributed agent systems and for communication between agents
- Support for *teams* of agents (not used in this project)
- An integrated development environment incorporating drag-and-drop construction of agents from capabilities and plans (however, plan bodies are textual)
- A design tool for visualising the structure of an agent system

The concepts that JACK adds to Java are:

- **Agents:** An agent has *capabilities* (things it can do) and *beliefs* (information), it handles certain *events* by using *plans*.
- **Capabilities:** A capability is, in essence, a wrapper around plans, events, beliefs, and sub-capabilities. Capabilities are analogous to modules in that they are a mechanism for structuring a large system.
- **Belief sets:** A belief set is similar to a relation in a relational database. It can be used to store an agent's knowledge and state.
- **Events:** An event is central to the execution mechanism of JACK (and of other BDI agent systems). An event is posted when something happens and triggers plans.
- **Plans:** A plan is what the agent uses to do things. A plan consists of (i) an event type that will trigger it, (ii) a context condition that indicates when the plan is applicable, and (iii) a plan body that is executed when the plan is selected. The context condition is a logical condition that evaluates to true or false. The plan body is code written in a superset of Java (i.e. including JACK constructs).

JACK's execution model is based around events and plans. For each event, there is a number of plans that can be triggered by that event type. This set of plans is the *relevant* plan-set. When an event is posted the agent considers the relevant plan-set. For each plan in the relevant plan-set, that plan's context condition is evaluated. If the context condition evaluates to false then the plan is ignored and the next plan is considered; otherwise the plan is executed.

JACK provides a number of event types, with differing properties. A simple `Event` can only trigger plans within the agent that generated it, whereas a `MessageEvent` arrives from outside the agent (perhaps from another agent, or external data source) and provides a reply mechanism. Together these provide simple event driven (or reactive) behaviour. A further group of event types provides meta-reasoning about plan selection and recovery from plan failure.

If the posting event is[3] a `BDIGoalEvent` then failure will be handled by trying alternative plans. Only if all the relevant plans for an event have been tried will the event (and hence its parent plan) fail.

Another type of event that is handled differently is an `InferenceGoalEvent`. An event type that extends `InferenceGoalEvent` will run *all* applicable plans, rather than just the first one. For example, suppose we have the following plans, that all handle the same event *e*:

Plan name	Context condition	Plan body
plan1	*p(a)*	print(plan1);
plan2	*p(b)*	print(plan2); false;
plan3	*p(b)*	print(plan3);
plan4	*p(b)*	print(plan4);

Assume that *p(a)* is false and *p(b)* is true; then execution will proceed as follows:

- Case 1, *e* is a normal event:
  1. plan1 is considered - since the context condition is false, it is ignored.
  2. plan2 is considered - since the context condition is true, it is executed.
  3. plan2 is executed, this prints "plan2" and then fails.
  4. Since *e* is a normal event, the plan that posted it fails.

[2] http://www.agent-software.com

[3] Actually, if it *extends* `BDIGoalEvent`.

- Case 2, e is a `BDIGoalEvent`:
  1. plan1 is considered - since the context condition is false, it is ignored.
  2. plan2 is considered - since the context condition is true, it is executed.
  3. plan2 is executed, this prints "plan2" and then fails.
  4. Since e is a `BDIGoalEvent`, alternative plans are considered.
  5. plan3 is considered - since the context condition is true, it is executed.
  6. plan3 is executed, this prints "plan3" and then succeeds.
- Case 3, e is an `InferenceGoalEvent`:
  1. Since e is an `InferenceGoalEvent`, *all* plans are considered.
  2. plan 1 is considered but ignored (since the context condition is false)
  3. plan 2 is considered and executed, printing "plan2" (and then failing)
  4. plan 3 is considered and executed, printing "plan3"
  5. plan 4 is considered and executed, printing "plan4"

The `BDIGoalEvent`s provide behaviour which is standard in the BDI systems discussed earlier. `InferenceGoalEvent`s however provide a functionality more directed towards reasoning than acting. All relevant plans (or rules) are executed. This gives a behaviour similar to expert systems and was particularly useful for some of the inferencing needed in this system where one wants to draw conclusions across all possible members of a set (as opposed to taking a single course of action).

3 Application Domain Characteristics

There is a particular requirement for improved aviation forecasts, and an important component is the rapid amendment of forecasts as soon as the need for amendment is indicated. This may be achieved by continual comparison of weather conditions against forecasts, which would be labour intensive if done by humans. An automated alerting system can perform a continuous weather watch and ensure forecasters will be alerted to significant weather developments in real time so that amendments may be quickly issued. Less severe weather changes will also be alerted by the system. The quality and timeliness of current aviation forecasts will thus be continuously monitored and corrected. Similar mechanisms can be used to deliver these updated forecasts to a wider audience.

3.1 Many clients with different needs

As listed earlier, there are many external client markets for an automated, real-time meteorological warning system. However, our experimental prototype is focussed on the aviation sector. Even here, clients fall into groups with different information requirements: forecasters themselves, regulatory authorities (Air Services Australia), commercial airlines (passenger and freight), military aviation and general (private) aviation.

Providing targetted information to multiple forecasters is one thing, but the aviation market may be much larger. The Australian domestic passenger fleet is under 1000 aircraft but provides over 500,000 domestic (plus 100,000 international) passenger flights (aircraft movements) per annum, whereas the general aviation fleet is over 10,000 aircraft flying for around 2,000,000 hours per annum (Civil Aviation Safety Authority 2002).

Figure 1: A typical automatic weather station.

```
TAF YMML 122218Z 0024
24006KT 9999 FEW025 BKN030
FM02 18015KT 9999 SCT040
FM17 25006KT 9999 BKN025
T 15 19 20 16 Q 1028 1026 1025 1026
```

Figure 2: An example of a TAF, a forecast of weather around an airport, encoding among other data the future temperature (T) and pressure changes (Q) on the last line.

3.2 Many data sources

There are also many existing sources of meteorological data currently available for generating weather alerts. These include

- raw observations provided by automatic weather stations (AWS, Fig. 1) available in 1 minute (instantaneous) and 10 minute (averaged) forms;
- filtered services such as METARS (a routine meteorological report issued every half-hour from particular stations, either automatically or by human);
- localised forecasts such as terminal aerodrome forecasts (TAF, highly abbreviated forecasts of weather around airports intended for pilots, Fig. 2);

- thunderstorm predictions from the TITAN (Thunderstorm Identification Tracking Analysis and Nowcasting) system, which produces short term (up to 2 hour) trajectory predictions as new radar data arrives every 5 or 10 minutes (Dixon & Wiener 1993);
- email notifications, e.g. from the Volcanic Ash Advisory Centre (VAAC);
- real-time lightning detection systems (e.g. LPATS[4], based on radio interference).
- direct observations (such as Clear Air Turbulence) from pilots en route;

3.3 Alerts

Alerts (or warnings) can be raised when inconsistencies are detected, either between a forecast and current observations, or between multiple observations or predictions from the same or comparable sources. When an inconsistency is found, the system can alert interested clients.

For example, an inconsistency between a TAF and corresponding AWS or METARS observations can be delivered as an alert to the current responsible forecaster for that region who may potentially change the TAFs issued in the future, thus leading to removal or lessening of the inconsistency. An intelligent system can compare these data streams and analyse them in various contexts: for instance, inconsistency, TAF not issued, TAF expired and TAF unrealistic. As part of such a system, intelligent agents can reason about such things as

- whether this alert has previously been issued,
- how important the alert is,
- whether the alerts are being responded to,
- which forecaster(s) to direct the alert to.

4 An Initial Implementation

The current prototype is an end-to-end demonstration of all the architectural capabilities required (subscriptions, data routing, communication with data sources, self-describing data, and simple service descriptions and service location mechanisms). While this example is relatively small, it provides a basic structure that can be used and refined for building the larger system.

As described in section 2, BDI agents provide a ready implementation vehicle for adaptive, distributed systems. Individual agents can be extended via additional plans to cope with new situations, and additional agents can be deployed to distribute the workload or provide new functionality. Ensuring accurate but minimal communication within the agent network then becomes the major problem.

4.1 A Pipe-and-Filter, Subscription Architecture

The architecture of the system contains a number of specially developed agents, a number of existing components, including the real-time data input system, and the data representation and management layer which is crucial to the overall architecture.

These components can all be run on the one machine or can be run on different machines across the network. In the pilot we have successfully run the system with components running on an operational server with real-time input data communications, a test system on a development machine, and agent driven graphical user interfaces (GUIs) running on forecaster workstations. See Fig 4.

The components are:

- Independent sources of AWS, TAF, VAAC and TITAN messages
- GUI instances that receive alerts and display them
- The main components of the system that receive TAF, AWS, VAAC and TITAN messages and issue alerts.

These components communicate using TCP/IP or JACK `MessageEvent`s, sending objects encoded using tree-table-xml (see Section 4.2) or serialized TTables contained in JACK messages.

The main components of the overall system are themselves agent systems that each contain a *DataStreamDispatcher* agent and some number of *Monitor* agents. The DataStreamDispatcher agent is responsible for managing incoming subscriptions and for routing messages. A TAFMonitor agent will subscribe to TAF and AWS messages and will generate discrepancy based alerts that it sends to the DataStreamDispatcher, which are then routed to the appropriate subscribers. Similarly, AbsAlert agents issue alerts when AWS values exceed their thresholds, VaacAlert agents issue volcanic ash alerts and the TitanAgent alters on changes in thunderstorm activity, based on alerts or messages from their respective sources, and distribute these alerts via their own DataStreamDispatchers. The internal structure of each agent component in the prototype is depicted in Fig 3.

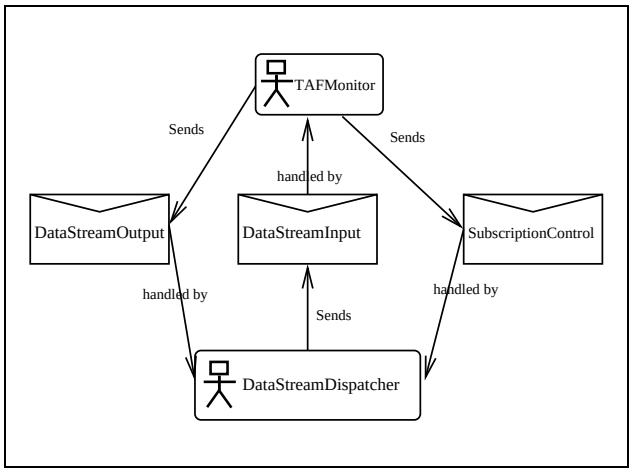

Figure 3: Structure of the agent network involved in subscription

Benefits

The pipe and filter design serves to minimise communication requirements, placing the relevant intelligent processing as close to the required information sources as possible. The subscription mechanism enables flexible decoupling of the producers and consumers of alerts. `InferenceGoalEvent`s are used by the DataStreamDispatchers to distribute messages to each subscriber, while `BDIGoalEvent`s are used by the Monitor agents to work through a range of alternative plans for alert generation.

[4] http://www.gpats.com.au

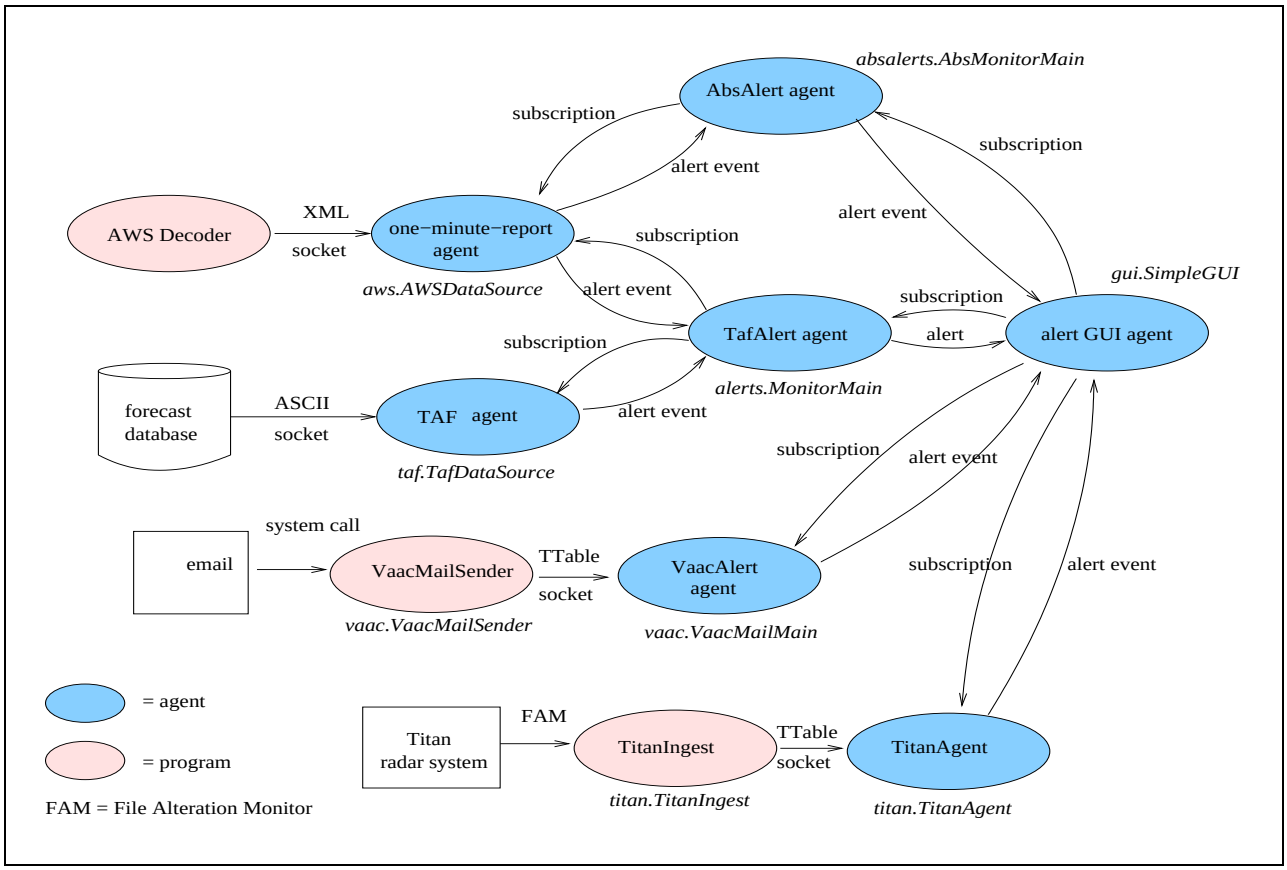

Figure 4: Diagram showing broad data-flow within alerts system.

4.2 Flexible Data Sharing via TTables

A generalised XML format known as tree-table-xml (Gorman et al. 2002) is under development in the Australian Bureau of Meteorology. Its design is intended to accommodate current and future meteorological XML format requirements by being extremely generic. Instead of representing meteorological metadata in XML tags, tree-table-xml defines a high-level meta-metadata structure called a `ttable`. This is not specific to meteorology, but is a generic format capable of handling a wide variety of data. This XML format also separates the metadata from the data. Its document type declaration (DTD) is shown below:

```
<!DOCTYPE tree-table-xml [
  <!ELEMENT tree-table-xml (ttable)>
  <!ELEMENT ttable (row*)>
  <!ATTLIST ttable name CDATA #IMPLIED>
  <!ELEMENT row (col+)>
  <!ELEMENT col (#PCDATA | ttable)*>
  <!ATTLIST col name CDATA #IMPLIED
                type CDATA #IMPLIED>]>
```

Note that the tree-table-xml DTD consists of just four meta-meta elements: `tree-table-xml` (the root element), `ttable`, `row` and `col`. Minimal attributes are defined for bootstrapping metadata: `type` and `name`. The message data is contained in a table called *data* and the corresponding metadata is contained in a related *metadata* table. Each column in the data table has a corresponding row in the metadata table. For instance:

data			
station_ name	**wind_ speed**	**wind_ direction**	**air_ pressure**
Melbourne	13.0	128	1001.0
Mildura	7.0	172	998.0
Avalon	20.0	117	1001.0

metadata			
element name	**unit**	**data type**	**significant digits**
station_name	-	string	0
wind_speed	knots	double	3
wind_direction	degrees	int	3
air_pressure	hectopascals	double	4

Benefits

This simple high-level design facilitates the development of software that can process a tree-table-xml document without knowing its content type. The TTable allows any kind of data to be expressed, including meteorological, service description metadata, system administration data, and agent oriented information. New data types may be introduced without impacting negatively on existing agents.

5 Evolving and extending the system

5.1 Adding Volcanic Ash Alerts

Partly as an exercise in determining how flexible the alerting system is, we put together an alert from a volcanic ash email advisory service (known as the Volcanic Ash Advisory Centre or VAAC). To this end, we have created a new email client and subscribed it to the volcanic email list. These emails were then piped into a transient Java process which scanned the email for the strings ‘volcan’, ‘erupt’ and ‘ash’ (not all emails to this list are actually about eruptions) and the name of any volcano in our region (from a database of volcano names and locations). If found, the system sends a TTable message to a JACK volcanic alerting agent, which in turn may trigger an alert.

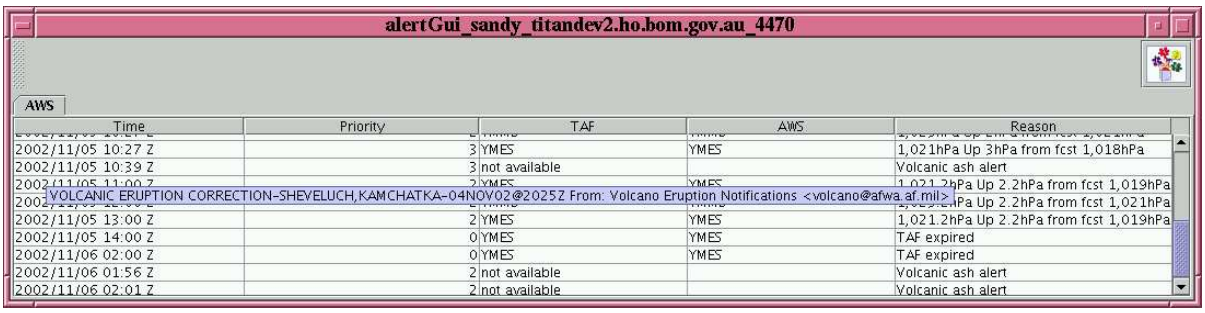

Figure 5: Example alert GUI showing volcanic ash mouse-over information

The volcanic ash alerting agent is available for any other JACK agent in our Bureau system to subscribe to (see Fig 4). These subscribers will usually be an alert GUI sitting on a forecaster's desk, see Fig 5. The alert contains the first 30 lines of the email, so it is available to the forecaster within the GUI to allow manual elimination of false positives.

This system extension took about 2 days to put together, demonstrating that our basic mechanism is simple, flexible and functional.

5.2 Adding TITAN Alerts

Following this, we also added an alerting agent for the TITAN thunderstorm prediction system (Dixon & Wiener 1993). TITAN uses radar to detect thunderstorms within a 200km radius - currently only Sydney and Melbourne - and tracks them using a consistent label. Storm data are kept in a file which is updated after every new radar scan (5 or 10 minutes). We hook into this system using FAM (file access monitor) which runs a shell script whenever files of interest change. This script initiates a Java job which converts the ASCII TITAN data file into a multi-row TTable, opens a socket connection to the machine with the TITAN JACK agent process running, and passes the TTable across.

The JACK agent process checks through the TTable finding new storms over certain thresholds, or old storms recently exceeding thresholds not already alerted, and if any are found, creates an alert TTable. This is then posted to all subscribers to this service.

The information posted on the alert GUI consists of the radar name, the storm number (so forecasters can identify it on external display systems), its location (lat-long and radar centric), and the storm parameters (cubic kilometres, kilotons of mass, speed, bearing, height in km, etc).

This extension was more complex than that for volcanic ash alerts, but took just over a week to add.

6 Deployment Experience

The alert system has had its first exposure to aviation forecasters, the alert GUI used can be seen in Fig 5. This provided valuable feedback on a number of issues, mostly around GUI look and feel, which we will address in the near future.

There were a number of deployment issues, broadly: self-healing from system failure and system evolvability.

6.1 Self-healing from system failure

To minimise system coupling, we have implemented the publish-subscribe pattern as noted above: when an agent subscribes to a service, it is granted a lease for a certain period. It then must resubscribe before that period has expired to continue getting the service. In this way, if a service is added or replaced by another, clients are able to seamlessly reconnect to the new service (assuming it has the same name). This has the added benefit of providing self-healing.

All distributed systems are vulnerable to failures in software, machines and networks, any one of which may potentially bring down the system. Manual intervention to fix failures is unrealistic and self-healing is necessary. In the publish-subscribe pattern, each server checks whether clients still have a valid lease before providing the service, and if not discards that client's subscription. On the client side, if a server fails the client will attempt to resubscribe until the service is again available. In this way the entire system self heals without immediate human intervention. This was amply demonstrated when, during a recent sustained network interruptions, the system promptly reestablished its internal connections.

6.2 System Evolvability

Software upgrades, updates and withdrawal of agents would also leads to system failure if this were not managed.

- The use of JACK facilitates easy implementation of new agent behaviour by adding new plans within a capability that are applicable in certain situations, adding new capabilities within an existing agent, or adding new agents to the system. For instance, the new subsystem which alerts on volcanic ash detections was implemented in less than two days.
- Leasing allows developers to withdraw and replace a component safely.
- Overriding the Java serialVersionUID on transmitted classes (to remove dependency on particular compilations of classes at either end of a message transmission via serialization) allows components commonly transmitted between machines to be extended and replaced incrementally and safely.
- The use of the generic data object TTable (see Section 4.2) and its externalized text format treetable-xml allows safe extension of data structures without recompilation.

The subscription model made the system very flexible, with alert GUIs running both on the forecaster's desk, and several displaying the same data on the development machine. The forecaster's GUI subscribed only to TAF alerts, whereas the development GUI subscribed to both TAF alerts and volcanic ash alerts. The subscriptions are controlled by drop-down menus on the GUI.

The forecasters now have access to the alert GUI via a menu option on their workstations, so we can easily expose them to future versions of the system simply by uploading new Java library files.

7 Planned Extensions

In addition to incorporating new data types and sources, for example lightning strike information, we now wish to evaluate the alerting system with additional clients.

One issue reported by the first set of aviation forecasters, was the inability to be more selective about which alerts were delivered. To date, the only configuration possible has been whether or not to subscribe to a particular type of alert agent. It was not possible to limit alerts by geographical region or selectively modify the preset thresholds. Regional forecasters will want a low threshold for alerts within their region, and higher thresholds further afield.

Similar configurability will be required when delivering alerts to aircraft en route. When flying from Melbourne to Perth, pilots will probably not be interested in thunderstorm alerts for Cairns, or ground level fog in Adelaide as they fly high overhead (although fog at Perth persisting until predicted arrival would be of interest), or weather alerts for regions they have already passed through. They may also want to specify different thresholds at different ranges and times.

Additional scenarios involve the introduction of new data sources and types, as yet unknown to potentially interested downstream clients. Examples include:

- a new localised severe weather alert of known type, predicted to intersect the current flight path,
- new types of weather alert service, e.g. the lightning service, becoming available along the flight path.

To minimise communication overheads, it would be best to push such selectivity as close as possible to the alerting sources, rather than receiving copious unwanted messages and then having to exclude them locally. This will require flexible and dynamic subscription configuration, rather than the static implementation in use at present. Both intelligent agents and human operators may be involved in this configuration process, which could require several exchanges (in effect a configuration protocol).

To date we have investigated the service discovery mechanisms provided for current client-server and peer-to-peer models, but these seem limited to exact matching on types and/or attributes (perhaps due to their reliance on distributed hash tables for implementation efficiency). What we will require is more along the lines of the run-time message filtering provided by the Java Messaging Service (Chappell & Monson-Haefel 2000, Sun Microsystems 2003). Further alternatives are listed in (Campailla et al. 2001).

Handling overlapping (and potentially conflicting) data resources provides further complexity. Simple duplications should be resolved as far upstream as possible, in order to minimise traffic; but conflicts may need to be propagated to a human operator for resolution. Service duplication will also impact recovery from partial network interruptions.

8 Conclusion

JACK and TTables have proven extremely effective in building and extending this experimental alerting system. The level of data abstraction provided by TTables plus the message passing provided by JACK has meant that the underlying communication and leasing infrastructure has not required modification to accommodate additional data types as new services are added. A new Monitor agent can be written, or new JACK plans or capabilities added to existing Monitor agents, in order to process the additional TTable data.

The pipe and filter architecture has served to minimise communication traffic, and provided a modular location for intelligent processing. Combined with the subscription mechanism, this modularity has also aided ready incorporation of new services implemented as new JACK agents. Several varieties of JACK reasoning (event handling) have been exploited in subscription and alert handling.

The subscription mechanism combined with leasing has also made the prototype more tolerant of system failures.

The main problem to date has been the lack of flexibility in the "all or nothing" subscription system. This will be the focus of the next stage of research and development.

References

Bratman, M.E. (1987), *Intentions, Plans, and Practical Reason*, Harvard University Press, Cambridge MA, USA.

Busetta, P., Rönnquist, R., Hodgson, A. & Lucas, A. (1999), *Jack Intelligent Agents - Components for Intelligent Agents in Java*, Technical Report 1, Agent Oriented Software Pty. Ltd, Melbourne, Australia.

Campailla, A., Chaki, S., Clarke, E.M., Jha, S. & Veith, H. (2001), Efficient Filtering in Publish-Subscribe Systems Using Binary Decision *in* 'Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001) Toronto, Ontario, Canada', pp. 443–452.

Chappell, D. & Monson-Haefel, R. (2000), *Java Message Service*, O'Reilly.

Civil Aviation Safety Authority, Australia (2002), Appendix A: CASA's Environment – Comparative Data, *in* CASA Corporate Plan, 2002-03 to 2004-05.

d'Inverno, M., Kinny, D., Luck, M. & Wooldridge M. (1998), Formal specification of dMARS, *in* 'Intelligent Agents IV: Proceedings of the Fourth International Workshop on Agent Theories, Architectures, and Languages (ATAL-97) Providence RI, USA', Lecture Notes in Computer Science **1365**, Springer Verlag, pp. 155–176.

Dixon, M. & Wiener G. (1993), 'TITAN: Thunderstorm Identification, Tracking, Analysis, and Nowcasting – A Radar-based Methodology', *Journal of Atmospheric and Oceanic Technology* **10**(6), 785–797.

Georgeff, M.P. & Ingrand, F.F. (1990), Real-time reasoning: the monitoring and control of spacecraft systems *in* 'Proceedings of the 6th IEEE Conference on Artificial Intelligence for Applications (CAIA-90) Santa Barbara CA, USA', pp. 198–204.

Georgeff, M.P. & Lansky, A.L. (1986), 'Procedural Knowledge', *Proceedings of the IEEE — Special Issue on Knowledge Representation* **74**(10), 1383–1398.

Gorman, M., Kelly, J., Ryan, C. & Sanders, C. (2002), Meteorological data and XML *in* 'Meeting of Expert Team on Data Representation and Codes, Prague, Czech Republic', Commission for Basic Systems, World Meteorological Organization. Available at http://www.wmo.ch/web/www/DPS/ET-DR-C-PRAGUE-02/Doc6(1).doc

Huber, M.J. (1999), JAM: A BDI-theoretic Mobile Agent Architecture, *in* 'Proceedings of the 3rd International Conference on Autonomous Agents (Agents'99) Seattle WA, USA', ACM Press, New York, USA, pp. 236–243.

Ingrand, F.F. & Georgeff, M.P. & Rao, A.S. (1992), 'An architecture for real-time reasoning and system control', *IEEE Expert* **7**(6), 34–44.

Ljungberg, M. & Lucas, A. (1992), The OASIS Air Traffic Management System, *in* 'Proceedings of the 2nd Pacific Rim International Conference on Artificial Intelligence (PRICAI'92) Seoul, Korea', pp. 236–243.

Sun Microsystems (2003), *Java Message Service Tutorial*, Sun Microsystems, Inc. http://java.sun.com/products/jms/tutorial.