A deployed multi agent system for meteorological alerts

Sandy Dance and Malcolm Gorman
Bureau of Meteorology
Melbourne, AUSTRALIA
{s.dance,m.gorman}@bom.gov.au

Lin Padgham and Michael Winikoff
RMIT University
Melbourne, AUSTRALIA
{linpa,winikoff}@cs.rmit.edu.au

ABSTRACT

The Australian Bureau of Meteorology has a requirement for complex and evolving systems to manage its weather forecasting, monitoring and alerts. This paper describes a system that monitors in real time the current terminal area forecasts (forecasts for areas around airports) and alerts forecasters to inconsistencies between these and observations obtained from automatic weather station (AWS) data. The contributions of the paper are a description of the overall architecture including legacy components, and the mechanisms that have been used to interface to legacy components; a description of an inferencing mechanism, available in recent versions of the JACK Intelligent Agents toolkit which has been particularly useful in some of the reasoning needed in this application; and a detailed description of the architecture for data sharing and data management. The system is currently deployed and a project is underway to extend this to a much larger system.

1. INTRODUCTION

The Australian Bureau of Meteorology[1] head-quartered 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 it is currently in the process of developing a sophisticated software system in which intelligent agents play a significant role.

There are a number of challenges to be met in developing this system:

- 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 system 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 co-operation. 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 agents and by legacy software.
- The system involves a range of complex goals, a highly dynamic environment and some complex inferencing.

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

Agents at Work: Deployed Applications of Autonomous Agents and Multiagent Systems, AAMAS'03 July 14-18, 2003, Melbourne, Australia.

This paper describes a pilot project which is the initial stage of this large and complex system. The contributions of this paper are a description of the overall architecture including legacy components, and the mechanisms that have been used to interface to legacy components; a description of an inferencing mechanism, available in recent versions of the JACK Intelligent Agents toolkit which has been particularly useful in some of the reasoning needed in this application; and a detailed description of the architecture for data sharing and data management. The pilot system has currently been deployed in the Bureau and a long term project has been approved and funded.

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. 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 goals that must be addressed by the system, such as detecting particular meteorological phenomenon, resolving inconsistencies in information, providing appropriate focused information to users, watching closely particular geographical areas (e.g. around airports), etc. This combination of a range of complex goals and a highly dynamic environment, makes a system incorporating intelligent agents, which can be both reactive and proactive, a natural choice.

The organization of this paper is as follows: in the first section we describe the overall requirements of the system, including that of the user interface. In section 3 we describe the overall architecture and address in particular the legacy components that were used and the generic tools that we used to interface with these. Section 4 provides a detailed description of the data sharing and management layer which is critical to the overall design and to the ability of the system to continue to evolve over time. Section 5 describes briefly the JACK Intelligent agents toolkit, and in particular introduces an inferencing feature not usually found in BDI (Belief, De-

```
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 1: An example of a TAF, a forecast of weather around an airport, encoding among other data the future temperature and pressure changes on the last line.

sire, Intention) agent systems, which is useful in doing some of the complex reasoning required by this system. Section 6 contains a brief description of alerting agent behaviour. Finally we conclude with information on deployment of the system and plans for future work.

2. APPLICATION DESCRIPTION

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.

The pilot system which is the focus of this paper monitors in real time the current terminal area forecasts (TAF, highly abbreviated forecasts of weather around airports intended for pilots, Fig 1) looking for inconsistencies between them and current airport observations as provided by automatic weather station (AWS, Fig 2) data available within the Bureau. When an inconsistency is found, the system issues an alert to the forecaster. Such an alert has the potential to cause forecasters to change the TAFs issued in the future, thus leading to removal or lessening of the inconsistency. Input to this system is the TAFs and AWS data obtained from the Bureau's current real-time communications system. The network of agents compares these data streams and analyzes them for various scenarios: for instance, inconsistency, TAF not issued, TAF expired and TAF unrealistic. The agent system reasons about such things as

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

The system 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 the scope is relatively small, the system provides a basic structure that can be used and refined for building the larger system.

The Australian Integrated Forecast System (AIFS) [1] is the hardware and software platform on which FSEP is being implemented.

Figure 2: Typical automatic weather station (AWS).

Forecasts are prepared and stored numerically to allow ready comparison with weather observations and forecast guidance. Guidance includes numerical weather model predictions that forecasters use to help guide their forecasting, as well as alerts to direct their attention to particular issues, such as inconsistencies or predicted weather patterns of concern.

2.1 Requirements for the System

2.1.1 User Interface

The Intelligent Alerts user interface is intended to reduce forecaster time and effort, so it is important to ensure that alerts are not excessive or intrusive. The user interface should be user friendly and intuitive with personal default settings, alert categories, alert priorities, manual and automated threshold setting, and easy access to alert details.

2.1.2 Graphic Alerts

Some alerts will be presented graphically, such as an overlay of the current (numerical) forecast with the model forecast (guidance) calculated. An unobtrusive arrival alert indicates to forecasters when a new model has been calculated, while a single discrepancy alert advises of all significant discrepancies between the forecast produced by the forecasters and that suggested by the system. The forecaster can then look at the graphical representation of the forecast, the guidance and the discrepancies between them.

2.1.3 Verification

Verification alerts provide *post hoc* feedback to forecasters. After the forecast period has elapsed and sufficient observational data has arrived, the accuracy of the forecast can be assessed. A systems approach to verification, providing immediate alerts to forecasters

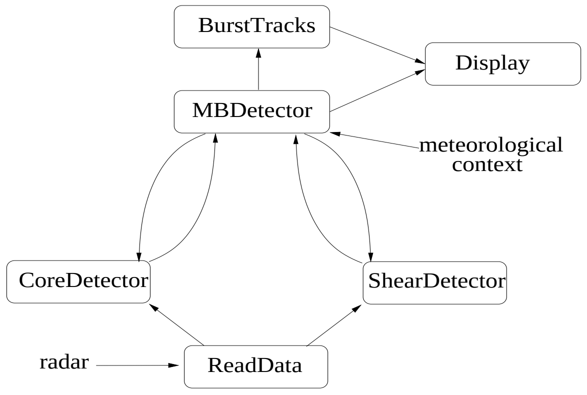


Figure 3: The network of agents used for microburst detection.

while the forecast is fresh in memory, provides the basis for continual improvement and higher forecaster satisfaction.

2.1.4 External Customers

Intelligent alerts can provide the opportunity to automate the notification to external customers of significant weather changes via email, event messaging, telephone, pager or PDA. The demand on forecasters and ancillary staff to respond to telephone queries at the height of a severe weather situation may thus be reduced, while providing a more timely and consistent real time alerting service to important clients.

2.1.5 An existing microburst agent system

An existing agent-based microburst detecting system [2] will be integrated into the alerting system described here. Microbursts are regions of high-shear and strong down-drafts associated with thunderstorms that are a hazard for aircraft, and can be detected with real-time weather radar. In the microburst system, the detection problem is broken down into a number of steps, each undertaken by its own agent. One agent seeks areas of high wind-shear in low-level radar sweeps, either autonomously, or as a result of inquiries from elsewhere in the system. Another seeks connected 3D regions of high radar reflectivity, indicating hail or high rainfall. These and other low-level agents alert a higher-level agent to the possibility of a microburst, which then looks for the signature of a microburst by sending queries to other lower level agents (see Fig 3). This system, although relatively simple, compares well with other more complex conventional systems [3, 4] in its ability to detect microbursts.

3. SYSTEM 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 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:

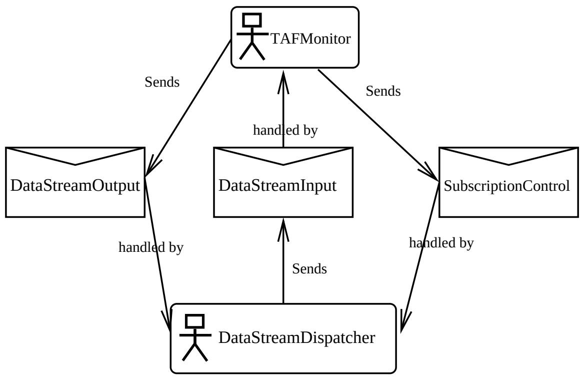


Figure 5: Structure of the agent network involved in subscription

- A source of TAF messages
- A source of AWS messages
- GUI instances that receive alerts and display them
- The main component of the system that receives TAF and AWS messages and issues alerts.

These components communicate using TCP/IP or JACK Intelligent Agents^{TM}[2] messages, and send objects encoded using tree-table-xml (see Section 4.1) and serialized TTables contained in JACK messages.

The main component of the system is an agent system that contains a *DataStreamDispatcher* agent and some number of *TAFMonitor* agents. The DataStreamDispatcher agent is responsible for managing subscriptions and for routing messages. A TAFMonitor agent will subscribe to TAF and AWS messages and will generate alerts that it sends to the DataStreamDispatcher, which are then routed to the appropriate subscribers. The structure of the agent component of the system is depicted in Fig 5.

3.1 Patterns

3.1.1 Publish-Subscribe

The agent network in the Intelligent Alerts system is connected using the Publish-Subscribe design pattern (see Ch5 in [5]). A server agent behaves as a publisher and advertises its service to other agents. Other agents may subscribe to the service and receive notification of events published by the server agent.

The benefits of the Publish-Subscribe pattern are:-

- Agents can be varied and reused independently without interfering with their respective subscribers or publishers.
- State changes in one agent can trigger state changes in other agents without knowing how many agents need to be changed.
- Agents can notify other agents without knowing about the other agents, and avoid tight coupling.

[2]JACK Intelligent Agents is a product of Agent Oriented Software who generously assisted us with this project. See http://www.agent-software.com.

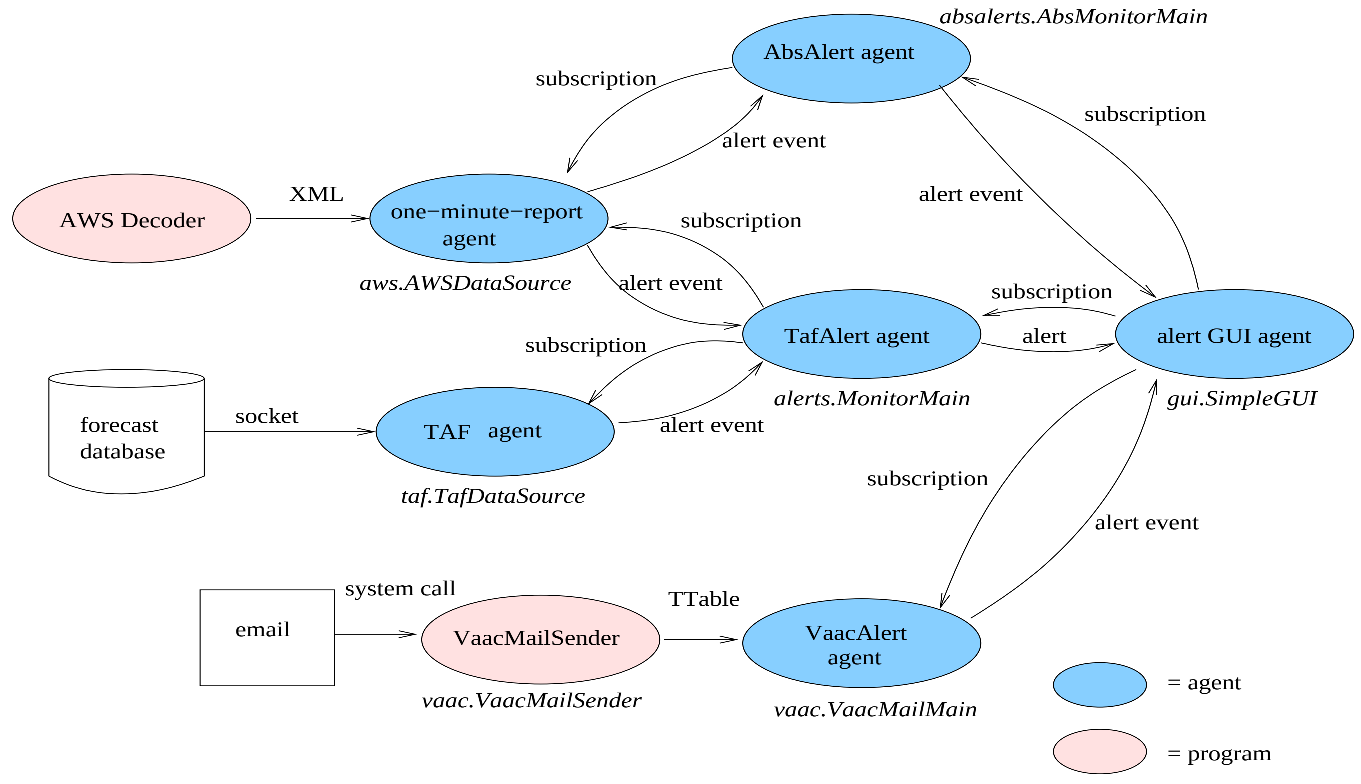


Figure 4: Diagram showing broad data-flow within alerts system.

3.1.2 Pipes and Filters

The overall architectural design of Intelligent Alerts is based on the Pipes and Filters Architectural Pattern (see Ch2.2 in [6]). The Pipes and Filters pattern provides pipes that allow a system to be structured and to pass data between adjacent filters. Intelligent Alerts implements pipes as one-way pipes using the Publish-Subscribe design pattern where a down-stream (client) agent actively subscribes to an up-stream (server) agent, and multiple down-stream agents may subscribe. This modification permits a network of agents to interact.

The benefits of the Pipes and Filters pattern are:-

- No intermediate files or database entries are necessary. All information may be stored and transmitted in generic transient event messages (see Section 4.1).
- Filters (agents) can be recombined to provide new behaviour. An event pipeline can enable rich, new behaviours [7].
- Support of recombination allows easy reuse of filter (agent) software components.

4. DATA SHARING

4.1 TTables

A generalized XML format known as tree-table-xml [8] 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 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

Each column in the `data` table has a corresponding row in the `metadata` table.

4.1.1 *General Processing Techniques*

The simple high-level design facilitates the development of software that can process a tree-table-xml document without knowing its content type. For instance, a Java class TTable has been defined that performs generic multi-key sorting and searching of tables. There is a Java TTableXml class that takes tree-table-xml documents as input and generates TTable objects as output. It does not need to know what kind of data is in the document. There is no limit to the number and type of metadata tables that can be defined for a data table. These may include any data that a downstream process may require to adequately use the document, such as a processing history. tree-table-xml is a canonical XML document format suitable for internal processing within a meteorological enterprise or for exchange of data with other meteorological agencies with shared software libraries for processing tree-table-xml. A tree-table-xml document can be used as a request for data. A client process can fill out a document with metadata describing the desired data and (if it is possible to fulfill the request) the document can be populated with a data table cell and returned to the requesting process. This lays the groundwork for the service discovery phase of this project.

4.1.2 *Standard Metadata Defined*

Metadata (and meta-metadata) are treated as data, and can therefore reside in a database (or a local XML copy of a central metadata repository) and be readily updated or extended. Agent metadata may also be standardized and shared between agents.

Event messages contain a set of TTable objects that are passed from agent to agent in the agent network, providing information to guide agent behaviour, and a vehicle for passing value added information to other agents. Each agent can interpret the data it receives by querying the accompanying metadata about element types and units, etc., shared by all agents in a metadata repository.

The TTable is a sophisticated form of the J2EE Value Object pattern [3]. 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. AGENT DEVELOPMENT TOOLKIT

The agent development toolkit which we used in this project was JACK Intelligent Agents [9].

JACK is a third generation agent system based around the concepts of Belief, Desire, and Intention (BDI) [10]. JACK is 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 visualizing 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.

If the posting event is[4] 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.

Yet 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:

[3]http://developer.java.sun.com/developer/restricted/patterns-/ValueObject.html

[4]Actually, if it *extends* `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 normal event, the plan that posted it fails.

- 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 InferenceGoal, *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 BDI systems such as PRS [11], JAM [12], dMars [13] and others. `InferenceGoalEvent`s however provide a functionality more directed toward reasoning than acting. All relevant plans (or rules) are executed. This gives a behaviour similar to expert systems and is particularly useful for some of the inferencing needed in this system where one wants to draw all possible conclusions (as opposed to taking a single course of action).

6. AGENT BEHAVIOUR

6.1 Floating threshold algorithm

To reduce the number of alerts to the minimum, we have produced an algorithm that only alerts upon significant change in the weather conditions (here pressure). This algorithm compares forecast pressure values with current observation values, and initially alerts when they diverge by more than 2 hectopascals (hPa).

This algorithm employs a ’floating’ threshold, which tracks the observed pressure value up and down, but only in steps of a ’margin’ value. Thus after an initial alert, the system only emits another alert when the observed value has increased by the margin over the previously alerted value. The algorithm for observations greater than the forecast:

```
initialize:  float = forecast + margin
upon new observation (obs):
     if obs > float
        generate alert
        float = obs + margin
     else if obs < float - margin
        float = obs + margin
        (unless less than initial float value)
```

Observations less than the forecast are similarly tracked with a separate lower float value.

This algorithm is intended to be as general as possible, so as to allow it to be used for many numerical forecasts/observations, ie, temperature, wind speed, visibility, and cloud ceiling. For values like wind direction, another algorithm will have to be developed to deal with values wrapping around 360 degrees. Another variant of the basic algorithm is used for ’absolute’ thresholds, where forecasters want to be alerted when wind speed, say, exceeds 25 knots regardless of the forecast (used by the AbsAlert agent in Fig 4).

6.2 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.

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

7. DEPLOYMENT ISSUES

The alert system has had its first exposure to aviation forecasters, the alert GUI used can be seen in Fig 6. This provided valuable feedback on a number of issues, mostly around GUI look and feel, which we will address in the near future.

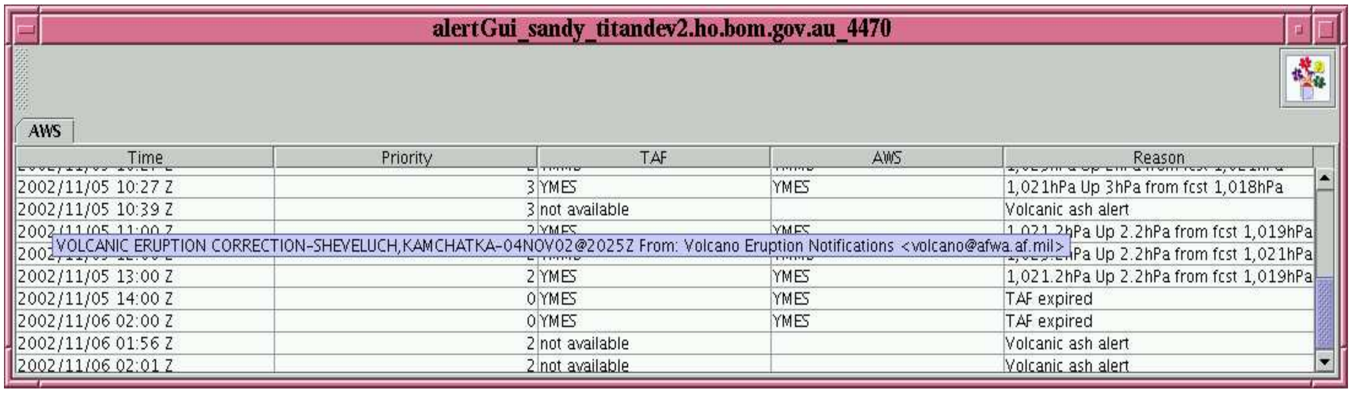
There were a number of deployment issues. These issue were broadly: flexibility, self-healing from system failure, and system evolvability.

7.1 Flexibility

To maximize flexibility, 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 (if it has the same name). See chapter 12 of [7] for more on leasing.

7.2 Self-healing from system failure

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, the server checks whether clients still have a valid lease before providing the service, and if not discards that clients subscription. On the client side, if a server fails the client will attempt to resubscribe until the



alertGui_sandy_titandev2.ho.bom.gov.au_4470

AWS

Time	Priority	TAF	AWS	Reason
2002/11/05 10:27 Z	3	YMES	YMES	1,021hPa Up 3hPa from fcst 1,018hPa
2002/11/05 10:39 Z	3	not available		Volcanic ash alert
2002/11/05 11:00 Z	2	YMES	YMES	1,021.2hPa Up 2.2hPa from fcst 1,019hPa
2002 [illegible]	[illegible]	[illegible]	[illegible]	[illegible]hPa Up 2.2hPa from fcst 1,021hPa
2002/11/05 13:00 Z	2	YMES	YMES	1,021.2hPa Up 2.2hPa from fcst 1,019hPa
2002/11/05 14:00 Z	0	YMES	YMES	TAF expired
2002/11/06 02:00 Z	0	YMES	YMES	TAF expired
2002/11/06 01:56 Z	2	not available		Volcanic ash alert
2002/11/06 02:01 Z	2	not available		Volcanic ash alert

VOLCANIC ERUPTION CORRECTION-SHEVELUCH,KAMCHATKA-04NOV02@2025Z From: Volcano Eruption Notifications <volcano@afwa.af.mil>

Figure 6: Example alert GUI showing volcanic ash mouse-over information

service is again available. In this way the entire system self heals without immediate human intervention.

7.3 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 in 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, a 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 4.1) and its externalized text format tree-table-xml allows safe extension of data structures without recompilation.

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

8. CONCLUSION AND FUTURE WORK

This paper has described an agent alerting system installed at the Australian Bureau of Meteorology. This system has evolved from an experimental agent system devoted to detecting microbursts, and is now at the pilot stage of providing alerts based upon discrepancies between forecast pressure and observations, and alerts from a volcanic ash alert mailing list. This pilot has given the authors confidence in the agent approach to system building in the Bureau context, as well as providing ideas for future work with this system. The subscription /leasing model employed to improve robustness and flexibility has also introduced a number of research issues for the future, namely: how will agents discover services that they need robustly and without human intervention, how will services advertise themselves to potential clients, and how to encode data so it describes its own semantics?

In the future we intend to expand the scope of the system so it alerts on an increasing variety of phenomena: for instance discrepancies between forecasts and observations for more data types like temperature and wind speed, serious weather events such as microbursts, storms, and hail, and system events like model availability, failed weather radar stations or communication links down.

This work will be carried out in a collaboration between the Bureau of Meteorology, the RMIT University Agents Group and Agent Oriented Software. This work is supported by an Australian Research Council Linkage Grant (grant LP0347025, “Open agent architectures for intelligent distributed decision making”).

9. REFERENCES

[1] Targett, P.S.: Predicting the future of the meteorologist: a forecaster’s view. Bulletin of the Australian Meteorological and Oceanographic Society **7** (1994) 46–52

[2] Dance, S., Potts, R.: Microburst detection using agent networks. Journal of atmospheric and oceanic technology **19** (2002) 646–653

[3] Albo, D.: Microburst detection using fuzzy logic. Technical report, National Center for Atmospheric Research, Boulder, Colorado (1994)

[4] Stoll, S.: Microburst detection by the low-level wind shear alert system. Weather **46** (1991) 334–347

[5] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Boston MA (1995)

[6] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture: A System of Patterns. John Wiley and Sons, Chichester (1996)

[7] Edwards, W.K.: core JINI. Prentice Hall, New Jersey (2001)

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

[9] Busetta, P., Rönnquist, R., Hodgson, A., Lucas, A.: Jack intelligent agents - components for intelligent agents in java. Technical report, Agent Oriented Software Pty. Ltd, Melbourne, Australia (1998)

[10] Rao, A.S., Georgeff, M.P.: An abstract architecture for rational agents. In Rich, C., Swartout, W., Nebel, B., eds.: Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning, San Mateo, CA, Morgan Kaufmann Publishers (1992) 439–449

[11] Georgeff, M.P., Lansky, A.L.: Procedural knowledge. Proceedings of the IEEE Special Issue on Knowledge Representation **74** (1986) 1383–1398

[12] Huber, M.J.: A BDI-theoretic mobile agent architecture. In: Proceedings of the Third International Conference on Autonomous Agents (Agents'99), Seattle, WA (1999)

[13] d'Inverno, M., Kinny, D., Luck, M., , Wooldridge, M.: A formal specification of dMARS. In Singh, M., Rao, A., Wooldridge, M., eds.: Intelligent Agents IV: Proceedings of the Fourth International Workshop on Agent Theories, Architectures, and Languages. Volume 1365., Springer-Verlag LNAI (1998) 155–176