Strathprints Institutional Repository

Gathering Experience in Trust-based Interactions

Colin English and Sotirios Terzis

University of Strathclyde
Department of Computer and Information Sciences
{Firstname.Lastname}@cis.strath.ac.uk

Abstract. Evidence based trust management, where automated decision making is supported through collection of evidence about the trustworthiness of entities from a variety of sources, has gained popularity in recent years. So far work in this area has primarily focussed on schemes for combining evidence from potentially unreliable sources (recommenders) with the aim of improving the quality of decision making. The large body of literature on reputation systems is testament to this. At the same time, little consideration has been given to the actual gathering of useful and detailed experiential evidence. Most proposed systems use quite simplistic representations for experiences, and mechanisms where high level feedback is provided by users. Consequently, these systems provide limited support for automated decision making. In this paper we build upon our previous work in trust-based interaction modelling and we present an interaction monitor that enables automated collection of detailed interaction evidence. The monitor is a prototype implementation of our generic interaction monitoring architecture that combines well understood rule engine and event management technology. This paper also describes a distributed file server scenario, in order to demonstrate our interaction model and monitor. Finally, the paper presents some preliminary results of a simulation-based evaluation of our monitor in the context of the distributed file server scenario.

1 Introduction

Trust management is emerging as a promising technology for facilitating collaboration with entities in environments where traditional security paradigms cannot be enforced due to lack of centralised control and incomplete knowledge of the environment. In particular, evidence-based trust management attempts to mitigate the risks inherent in interactions lacking concrete security assurances by gathering evidence to support trusting decision making.

Studying the literature on evidence based trust management highlights that most systems focus on sharing evidence and opinions among peers and combining this evidence to make trust decisions (e.g. [10]). However, the means to gather personal experiential evidence is often lacking. The systems that provide such functionality tend to use simple representations of experience, such as a numeric rating (e.g. [1]). To support an expressive trust model for decision making in complex interactions, it becomes more important to get detailed feedback upon which to base future decisions. Additionally, many systems rely on the user to provide feedback (e.g. [14]). Even in commercial systems, such as EBay the user provides very simple feedback ratings. However, many interaction decisions that might benefit from trust management techniques will take place in the absence of a user. Even with the user present, it may not be appropriate or convenient to require them to provide feedback. This introduces a requirement for feedback to be largely automated.

This work builds upon our earlier work in trust management [5], which defined a trust model [11] that recognises the strong link between personal observations and trust. The model views an interaction as a set of possible outcomes, based of a set of observable events within the interaction, which is organised in an event structure (see [11] for details). Computations can be defined over interaction outcome histories to derive a trust value for a specific entity. In conjunction with information on the costs of the possible outcomes, a trust value enables the evaluation of risk in an interaction to facilitate a decision process [13]. In [4], we defined an interaction model that extended this trust model to capture more detailed observations about generic interactions and their associated costs. An application developer can instantiate this model for a specific type of interaction to define a set of observations that may be made either directly or indirectly about the behaviour of a trustee. The observations are defined to represent the aspects of the interaction type that the developer deems relevant to a trusting decision. The model is event based, facilitating the automated gathering of objective evidence for subjective evaluation in a decision process. The same paper presented initial steps towards automating this evidence gathering, introducing a preliminary architecture for a generic monitor that could be used to follow interactions based on the model.

In this paper, we advance our previous work by refining the monitor architecture and examining the use of existing reactive technologies to provide an implementation of the architecture. After briefly outlining the interaction model in section 2, we describe the requirements for the monitor in section 3, followed by a refined monitor architecture in section 4. These refinements were the result of a more thorough investigation of the technologies used in the monitor and the instantiation of the model for specific application scenarios. In section 5 we present a prototype implementation of the architecture. In section 6 we present a preliminary evaluation of the prototype. This is based on a particular instantiation of the interaction model for a file server scenario (presented in section 6.1), which forms the basis for a simulation platform outlined in section 6.2. This simulation platform and the prototype monitor provide a basis for our evaluation, some preliminary results of which are presented in section 6.3. The paper concludes and looks to the future in section 7.

2 The Interaction Model

Our interaction model provides a number of extensions to the trust model in [11] that allow more detailed observations to be made about the state of an interaction in order to fully support a trust-based decision process. The focus of our model remains on observable events that capture a variety of aspects of the interaction. The events of the model can be further decomposed into a set of *trust events* (E_T) and a set of *cost events* (E_C). The trust events are those which capture the aspects of an interaction that reflect the trustee's behaviour in some way and hence something relevant to its trustworthiness (e.g. a file server's integrity). This set takes the form of an event structure. The cost events are added to the model in order to represent an occurrence of something that affects the costs associated with outcomes rather than the outcomes themselves. Thus we may also capture the dynamism of interaction progression from a cost perspective. To increase the flexibility of the trust events, we can further subdivide E_T into *directly observable events* and *quantified events*. While the former represent aspects of an interaction that can be expressed through single event instances, some aspects require the more abstract notion of quantified events, which are single logical events under which a series of low level observations can be aggregated, to form a *measure* of the quantified event (e.g. the latency measure of a file server). The low level observations are referred to as *measure events*. The relationship between the main event types can be seen in figure 1. Measure events are not shown, as although observable as single event instances, they are not part of the trust event structure, merely incorporated into an outcome via the measures of quantified events.

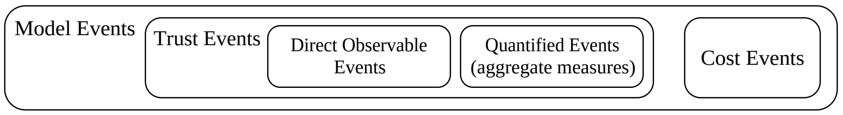

Fig. 1. Event Type Relationships.

Modelling a particular type of interaction involves identifying the above sets of observable events for the set of outcomes. Through these event sets, we have means of representing the state of an interaction from start to finish. Experience thus represents objective evidence which can be evaluated subjectively for trust decision making, such that we can differentiate fact and opinion. The feedback loses no information and can be as detailed as the specified interaction model. Furthermore, feedback can be provided throughout an interaction rather than at its conclusion, which can be useful in adjusting our trust opinions in a timely fashion. The model is described in more detail in [3].

3 The Monitor Requirements

The monitor is designed to operate as a service on a single device that can monitor interactions for a number of that device's applications. We have chosen to co-locate the monitor with the client application as this provides some privacy within the context of a single user's machine. In cases where the monitor needs to run on a resource constrained device, it could run in a proxy configuration providing access to an external service on a more powerful machine, in a manner transparent to the application.

From the interaction model described above, the natural separation of trust and cost events provides us with a means of separately representing the *trust-state* and *cost-state* of an interaction. All events from E_T seen so far in an interaction represent its trust-state. The cost-state is represented by a mapping from the possible outcomes to the currently associated costs and the set of events from E_C seen so far. Together, these states give us the *interaction-state*, which can be used to provide detailed feedback at any point during an interaction. The two main issues that arise for the monitor here are how to collect the events that contribute to the interaction state and when to pass the collected information back to the client application.

3.1 Event Communication

Even though the monitor and application may exist on the same machine, it is likely that the sources of many events will be remote. As we may have event sources external to both the monitor and client application, we can see three categories of interaction model events emerge. These are events internal to the client application, those external to the application yet from the trustee and those external to both, perhaps from some separate service. Event (or messaging) systems provide a form of asynchronous messaging that allows the monitor to be decoupled from event sources, while supporting many different event systems via plug-ins and maintaining the generic qualities of the monitor.

Features of individual event systems vary widely. A major factor is the connection model communication is based on. A single central event broker is inappropriate for distributed and heterogeneous event sources and prone to failure. Hierarchically distributed event servers or an even more decentralised peer-to-peer (P2P) network is preferable. Most event systems support both *point-to-point* (PTP) or *publish/subscribe* (pub/sub) messaging. PTP messaging uses dedicated queues and is suitable for internal client and trustee events, where the client can inform the monitor on how to subscribe to the particular source. However, events from an external service will generally be global rather than interaction specific in scope, such as cost events that affect more than one application. These events fit more appropriately with a pub/sub paradigm which allows for many-to-many event communication where the the subscriber does not need to know the specific source. Whether PTP or pub/sub is used, the client application must tell the monitor how and where to subscribe to event sources.

Our event system requirements also include the ability for the source to push events to a consumer rather than have the consumer periodically request notifications, in order that they be received in a timely fashion. The event notifications must permit parameters to incorporate any pertinent information about the event itself. Reliability of event notification is closely linked to the fault tolerance of the underlying connection model and its network protocols. However, many systems also offer some form of message persistence like store and forward or polling. Best effort delivery is likely to be the limit of our reliability guarantee for the monitor in general, as the monitor has little or no control over the measures the sources employ. Finally, security measures are important to ensure the privacy of generated events.

3.2 Provision of Feedback

Through the interaction model and a suitable event system, we have the means to both gather and represent interaction-state. We therefore need some means to reason over it in order to provide relevant feedback when required. At the end of an interaction, it is clear that we should pass all collected evidence to the application in order to update the interaction history with the new outcome and provide the best support for decision making possible. However, there may be some scenarios in which we would like either periodic updates during an interaction, or even notification of certain states that we deem important in order to take action and minimise damage. Furthermore, prompt feedback spanning across the ongoing interactions can provide the most up to date relevant evidence for decisions. We need to be able to communicate to the monitor the desired feedback and when to deliver it. Therefore, we use a rule-engine-based architecture whereby the application can specify rules about feedback and the monitor can reason over the state to meet the needs of a particular client.

We use a particular type of rule, called *Event Condition Action Rules* (ECA rules). ECA rule engines have been commonly used in active databases [12] for a number of years, but are now coming into more widespread use through decoupling them from specific database systems to make more generic reasoning engines [2]. ECA rules express an action to execute when some combination of events is witnessed, provided that a boolean condition holds true. These rules are the natural choice given that we have an event based interaction model to provide triggers for the rules and values from event counts, parameters and outcome costs for use in conditions. The client can define a set of these rules and communicate them to the monitor, such that it can provide feedback as defined in the action part of a rule. It should be clear that the monitor therefore need not understand the semantics of the events, rather just match patterns as defined in the rules. It is up to the definer of a specific rule to specify meaningful feedback for itself, which might take the form of a pertinent message or reports of current interaction-state.

The features of different existing ECA engines vary a great deal. A good framework for evaluating the range of functionalities can be found in [12]. For our purposes, it is mainly important to ensure that the engine provides for the kind of rules that are useful given the interaction model. For example, the operators permissible for event combinations and conditions is important. This will include primarily logical set combinations for events, and arithmetic functions and comparisons for the value based conditions. The engine must support parameterised events such that the associated values can be of use in conditions. The primary consideration for rule actions is that the communication mechanism for reporting feedback can be called upon. The mechanism may be either synchronous or asynchronous, as we can use the same inter-process mechanism used for the monitor's management API to provide callbacks, or use the event communication mechanism to send a feedback message. A further useful feature would be the ability to have rules generate events upon which other rules could be triggered, commonly referred to as the *cycle policy* of the engine. This would allow the monitor itself to generate events for chaining or even blocking other rules as specified by the rule set definer. In section 5 we will describe how our prototype monitor incorporates the functionality outlined here.

4 The Monitor Architecture

With the above considerations in mind, we can go on to describe an architecture for a monitor, which operates as a self-contained software component to enable it to be used in a generic fashion across a range of applications. In [4], we described our preliminary architecture, which has now been refined. The refined high-level architecture can be seen in figure 2.

The architecture highlights the responsibilities of the different components and the interfaces between them and the management API of the monitor. In this section we will highlight the major changes from the previous architecture in [4]. The monitor is itself constructed from various components. Firstly, an Event Manager (EM) is respon-

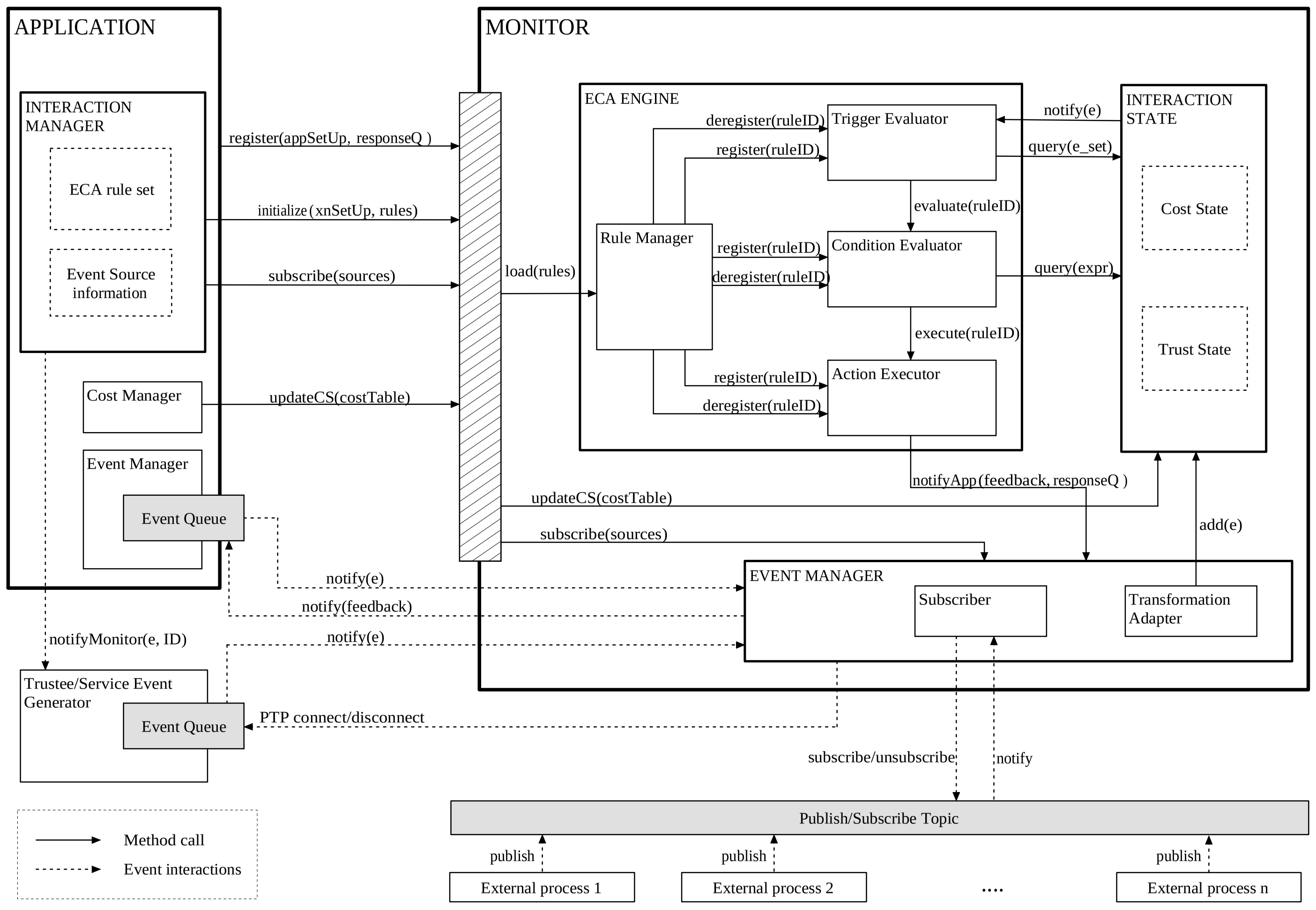

Fig. 2. The Monitor Architecture.

sible for subscribing to all events. The EM has been refined to clarify its responsibility for translating events for further processing. The Interaction-state Store (IS) component maintains trust and cost-state in working memory, accessible to the final major component, the ECA Rule Engine (RE), which has been refined based on investigations into rule engine implementations. These monitor components are not visible to the application, which interacts with the monitor mainly via its public API for management purposes.

The monitor API has been made more concrete in the new architecture, as can be seen in figure 2. The API provides the means by which an application can register itself and its interaction model, via the `register(appSetUp, responseQ)` method. To initiate monitor ing of a particular interaction, the `initialize(xnSetUp, rules)` method can be called. Via the `updateCS(CostTable)` method, an application can explicitly define costs to associate with an interaction's outcomes, perhaps triggered by feedback when the monitor receives a cost event. This is appropriate when the application has some complex means of determining what certain events mean in terms of costs it will incur. However, in many cases, the application will be abe to define a set of rules for cost updates which incorporate functions to alter the costs directly on the monitor.

The EM exposes a `subscribe(sources)` method for the monitor to forward subscription details from the API's `subscribe(sources)` method. The Subscriber subcomponent of the EM is responsible for creating and maintaining PTP and pub/sub connections to event sources based on this information. All event notifications are timestamped [1] and passed to the Transformation Adapter (TA), which then translates them from the source format into that used in the IS and RE. The Subscriber and TA can be extended via plug-ins for new event systems. The EM also provides feedback through the messaging system when the RE calls the `notifyApp(feedback, responseQ)` method as a rule action. The EM calls the IS's `add(e)` method to pass all event notifications into storage.

The IS represents the working memory for the RE, providing access to the relevant elements of interaction-state for rules. Sets of events and cost tables are stored and updated for each ongoing interaction. When an interaction's final outcome has been fed back to the application, the events of that outcome may be removed from the IS. The `updateCS(costTable)` method provides the means for the costs of a specific interaction's outcomes to be updated from the application. The methods `query(e_set)` and `query(expr)` allow the components of the RE to query the state of particular interactions for rule evaluation. Finally, the IS is also responsible for notifying the RE of new events via its `notify(e)` method.

Finally, we come to the reactive component of the monitor, the ECA Rule Engine (RE), consisting of a number of decoupled sub-components. The Rule Manager (RM) manages rule activation, execution, scheduling and represents the state of each active rule. Rules are registered via the `load(rules)` method and propagated to the relevant component. The RM maintains state for the event trigger part of rules, to be processed and updated by the Trigger Evaluator (TE) as notifications of events are re-

[1] The notifications are timestamped in the EM when observed as remote clocks cannot be relied upon for consistent time.

ceived through its `notify(e)` method. Trigger state can be seen as a tree structure of logical primitive event combinations, the expressiveness of which depends on the rule language. When the root of the tree becomes true, the `evaluate(ruleID)` method of the Condition Evaluator (CE) is called automatically. The CE is then responsible for evaluating the boolean expression in the condition part of the rule by querying the IS. Again the operators supported in the boolean expression depends on the rule language. If the condition is true, the CE calls the `execute(ruleID)` method of the Action Executor (AE), which schedules relevant rule's action for execution. The `notifyApp(feedback, responseQ)` method of the EM is commonly called by the action of a rule to pass relevant feedback as message through the appropriate event system. The message is defined in the rule and should contain all pertinent information.

The application component in figure 2 just outlines the type of components that the application might have, but this will be up to the developer of the application.

5 A Prototype Monitor

A prototype monitor has been developed for the evaluation of the architecture and interaction model. It was implemented in Java to take advantage of available class libraries and technologies for the two main functional aspects of the monitor; the ECA rule engine and the event system. The monitor exports an RMI remote interface for the application to interaction with, in accordance with the monitor architecture. The monitor runs as a single thread which enables single-point logging of interaction events as seen by the event manager, for the purpose of interaction trace replay and testing of different rule sets.

A myriad of rule engines are available (e.g. [7, 9]), which offer varied functionality. Having examined a range of such technologies, we chose the Java Expert System Shell (Jess) [6] to implement the ECA rule engine as it is very well documented and supported, with a large user community. It provides very good integration with Java, ranging from applications written purely in Java code to mainly Jess code simply launched through a Java application. The Jess Shell can also be used as a Java scripting environment to aid in rapid prototyping. As everything we reason over must be available in working memory, we have in effect combined the IS and RE into one component, the *ECA Engine*, which instantiates the Jess inference engine. Jess uses the Rete algorithm [6], in which rules have state modelled internally in a manner similar to the complex event tree mentioned in section 4. A Rete network is built from single input fact nodes and two input join nodes. The fact nodes represent patterns and the join nodes represent a number of *conditional elements* such as logical combinations of facts. Rete shares nodes across the set of rules for more efficient processing. A rule is executed once for each matching set of facts. Furthermore, queries over working memory can be defined and run under direct program control to process collections of facts. Jess is not explicitly designed for ECA rules, rather for inference in applications such as Expert Systems. However, ECA rules can be modelled by asserting facts for events and specifying conditions using *test* conditional elements that may contain arbitrary boolean expressions written in the Jess language. Furthermore, it is possible to define functions in Jess code to extend the functionality/operator set. Similar functions can be defined to perform the necessary actions, expressed on the left hand side of the rules.

5.1 Coding the Generic Interaction Model in Jess

Before we can assert facts into working memory to represent interaction-state, we must define templates for the facts in accordance with our interaction model. Thus the generic interaction model is defined in the monitor by running a batch file of template definitions through the instantiated Jess engine. An *interaction* template is defined with fields to store interaction, application and trustee IDs along with the queue-name to which feedback should be sent. A *measure* template allows for temporary storage of measure events until they are assigned to the relevant quantified events at the end of an interaction. An *outcome-cost* template allows a cost to be associated with a specific outcome for a specific interaction. An *event* template is defined with fields to link asserted event facts to a specific interaction and store a timestamp. This basic event template is further extended to a *trust-event* template and *cost-event* template, which themselves are further extended to produce *measure-event*, *direct-observable-event*, *quantified-event* templates etc. This extension mechanism is also the means by which a specific application's interaction model is defined. New specific event templates extend the basic event templates to give a hierarchy of event types (see the examples in section 6.1). This enables useful queries to be expressed over the working memory that allow, for example, all events relating to a particular trustee to be processed in some way. We have defined a basic set of queries over the templates of the generic interaction model, including queries over specific application templates based on template names.

```
(defquery find-xn-named-events
  (declare (variables ?xn ?name))
  ?ev <- (event (xnID ?xn))
  (test (eq ((?ev getDeftemplate) getBaseName) ?name)))
```

The results of queries can be iterated over and processed to provide useful functionality for test clauses (i.e. rule conditions). We therefore further define a set of generic functions that may be used in test clauses or for feedback actions. This set includes functions to determine the maximum/minimum/average/total value of a particular field from a collection of facts and functions to extract particular facts or values amongst others. An important function *notify-xn-app* is defined for rule actions, which calls a java method in the monitor from the Jess engine to return a feedback string via the response queue for a specified interaction.

```
(deffunction notify-xn-app (?xnID ?string)
  ((fetch MONITOR) notifyApp
    (get-specific-interaction-response-queue ?xnID) ?string))
```

In effect these queries and functions extend the expressivity of the rule language, so to be of use they must be defined in the working memory before rules can be defined that use them. In fact, a developer can also define new functions for use in test clauses, to be supplied as part of the interaction set up phase.

5.2 Implementing the Event Manager

The other major component for the monitor is the Event Manager, implemented to enable plug-in event systems via an extension of a MessengerLayer interface. For the monitor prototype, we wished to use a decentralised system, preferably with a P2P connection model. Furthermore, we felt a standards based approach that supported both PTP and pub/sub would be beneficial, to show that a number of implementations of such a standard could easily be supported. For these reasons, we sought an appropriate implementation of the Java Messaging Service (JMS)interfaces which provide a standard for asynchronous event communication in the Java language. Various implementations (or *JMS providers*) available which support a P2P connection model, but we decided to use MantaRay [8] as it provides its P2P functionality through a self-contained transport layer implemented over either TCP or SSH and HTTP, with the necessary discovery protocols. Furthermore, it implements persistence through a store and forward mechanism and has highly configurable logging mechanisms.

The plug-in built for MantaRay sets up the necessary JMS connections to the transport layer for both queue (PTP) and topic (pub/sub) based messaging. It manages collections of message senders, receivers, publishers and subscribers and facilitates the sending and publishing of events via method calls. Finally it acts as a message listener, passing any notifications onto the Transformation Adapter component. The messages received must be in the form of JMS Map messages in the prototype as these enable name-value mapping for event parameters. This is to simplify the Transformation Adapter of the plug-in to have only one input format which it must translate into Jess assertions. To enable translation, the application must supply the adapter with template objects for all event types in its interaction model. These templates define which event parameters are strings or numbers, ensuring correct assertion strings can be built for execution in the Jess engine.

6 Monitor Evaluation

Before presenting the evaluation of the monitor and interaction model, it is important to discuss the performance of the Jess and MantaRay technologies that form the basis of the monitor. As the Rete algorithm upon which Jess is based maintains rule state and only updates changes, its complexity is something like $O(R'F'^{P'})$ where R' is a number less than R the current number of rules, F' is the number of facts that have changed and P' is a number between 1 and the average number of patterns per rule [6]. Furthermore, the performance of a Rete-based system also depends on the number of partial matches generated by the rules, so badly written rules may exhibit poor performance. Performance of join computations for each rule can be tweaked to trade off memory usage against speed of computation. The usual messaging system performance tests include scalability for destinations (topics and queues), publishers and subscribers in terms of number of messages per second throughput. MantaRay's P2P architecture removes the concern over destination load as the destinations can be hosted on individual peers. The performance of MantaRay depends on logging levels, persistence mechanism choice and transport layer protocol. MantaRay supports database or file persistence, with file persistence offering far superior throughput. Using file persistence in a single queue or topic, with minimal logging and TCP connections, message rates between 6000 to 7000 per second have been reported in online discussion fora. It is also worth mentioning that MantaRay is a lightweight solution; the transport layer may need only 3 MB hard disk space.

Based on the above performance analysis, it is evident that these technologies, while suitable for deployment on laptop devices, are too heavy-weight for resource constrained devices such as PDAs. This could however be overcome using a proxy configuration to a more powerful machine. As the event sources can implement the queues and (for this scenario at least) the event notification rate is likely to be well within the above bounds, scalability is no problem for MantaRay. Furthermore, (at least with sensible rules) Jess can provide prompt feedback.

For evaluation of our monitor and interaction model we decided to follow a simulation based approach, as this provides the necessary control over the environment for varied experimentation. Simulation also has the benefit of being able to run a number of experiments in a short time and removes the possibility of incurring real world damage from running tests on a real implementation. Our approach involves simulation of a number of application scenarios. So far we have concentrated on a distributed file server scenario which we will outline below.

6.1 File Server Scenario

In our file server scenario, many users can subscribe to host files on many different distributed file servers for a specific duration. This is an interesting scenario from the point of view of the client trusting the server, as there is a rich set of clearly defined aspects of server behaviour that can be witnessed and the interaction has a duration that allows for continuous feedback. Furthermore, a number of interactions may be ongoing at any one point in time (even with the same server), thus prompt feedback from one interaction may be useful for decisions on others. First we define the trust events that reflect aspects of server behaviour in terms of the outcome of an interaction. The specific aspects used will depend on what the application developer deems important for decision making. The aspects we have chosen are the *availability* of the server, the *latency* of access, how well it protects the *integrity* of the hosted file, and how well it maintains the *confidentiality* of the file.

From this set of aspects, the developer can define the set of events that represent an outcome. The availability and latency aspects require quantified events to be defined, with an associated measure, as the individual measure events that reflect these aspects can be repeated. We model the other aspects as direct observable events. For example, we assume that the integrity of a file is either maintained throughout or not, although a different view could have been taken here to incorporate degrees of damage. We assume that we can only directly observe a breach of confidentiality, as we can never say for sure that confidentiality was maintained. Furthermore, we model whether the interaction lasted the full duration or the file was removed early. We thus define the following trust event and cost event types by extending the basic interaction model event types from section 5:

– **availability-qe** with associated measure events **available-me** and **unavailable-me** one of which is seen for any attempt to access the server.

– **latency-qe** with associated measure event **latency-me** with a *latency parameter*.
– **integrity-undermined-event** and **integrity-maintained-event** are conflicting; one must always occur in an outcome.
– **confidentiality-breached-event** may or may not occur for any specific interaction.
– **host-event** and **not-host-event** are conflicting; no other events are seen if not-host-event is received for an interaction.
– **bad-xnend-event** and **good-xnend-event** are conflicting; one must be seen to signify the end of an interaction.
– **cxn-cost-changed-event** has a *cxn-cost-change parameter* to represent the change in connection costs when changing, for example, from a broadband connection to dail-up.
– **file-update-event** has a *file-value-change parameter* to show the effects, for example, of updating a file with critical data.

As a Jess code example, consider the following which shows the extension of a generic cost event to give the `file-value-event` template:

```
(deftemplate cost-event extends event
  (slot appID (default "GLOBAL"))
  (slot xnID (default "GLOBAL"))
  (slot trusteeID (default "GLOBAL")))

(deftemplate file-update-event extends cost-event
  (slot file-value-change))
```

The event definitions described here, which capture the interaction model for an application, are passed to the monitor when the application registers with the monitor, via a batch file of Jess code. This file also contains other set up information, including the definition of an **fs-interaction** fact that extends the basic interaction fact to include a fileID and file-value, and **latency-measure** and **availability-measure** facts that extend the basic measure fact to link the relevant measure events to the relevant quantified events. The final part of this file contains rules triggered on the cost events to update the outcome costs. Once this file has run, a rule set for feedback can be defined.

6.2 The Simulation

Based on the file server scenario described in section 6.1, we have developed a simulation environment to provide an experimental platform for the evaluation of the monitor prototype. The simulation (outlined in more detail in [3]) comprises an application component and a file server component that can be instantiated on a number of remote machines, to provide a realistic environment in which to test the monitor and interaction model. Both the application component and the file servers are JMS enabled through individual Mantaray transport layers. Queues are used for communicating most events for privacy, with only the global `cxn-cost-changed-events` passed via topics. The file server exports an RMI interface for file hosting and subsequent file operations from which server behaviour can be observed (see figure 3).

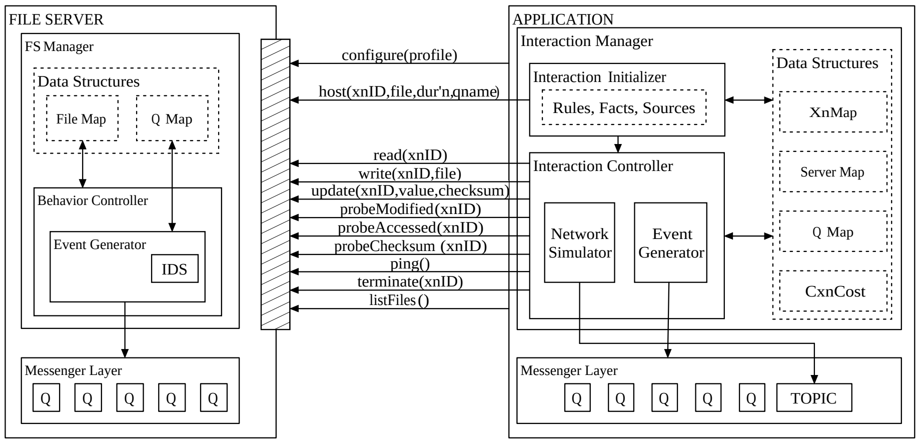

Fig. 3. The File Server and Application.

In order to keep the file servers lightweight for the simulation, we define file objects rather than create real files for storage. These encapsulate the file's ID, a file value, an integrity checksum and last modified and accessed timestamps. File servers store these file objects and perform a number of actions upon them based on a *behavioural profile* configured through the interface. This profile influences server behaviour on each aspect of latency, availability, integrity and confidentiality. The *Behavioural Controller* component uses these profiles to determine thresholds for certain behaviours, continually iterating over the currently hosted files and acting upon them in a number of ways. This includes generating `confidentiality-breach-events` and `integrity-undermined-events` for an interaction via the *Event Generator*, simulating an intrusion detection system (IDS) and disk corruption notifications. The Behaviour Controller can also alter a file's checksum, value or timestamps without generating an event, leaving the application to discover faults as a result of later method calls. Further to this, latency profiles influence the amount of delay added to operations to simulate a slow or overloaded server and availability profiles influence the chance of an `UnavailableException` being thrown by remote methods.

All the events defined in the file server interaction model may also be generated by the application based on the results of file operations when compared to a local file store. Once the application's *Interaction Initialiser* has hosted a file and initiated monitoring for the interaction, the *Interaction Controller* is responsible for iteratively executing file actions in the set of current interactions. To enable events to be generated from file operations, it provides wrapper methods around each of the remote method calls such that the relevant return values can be compared to the local store. The relevant JMS map messages are generated and sent to the monitor queue. Common wrapper functionality includes recording the time delay for a method call in order to generate a `latency-me` from any successful access. This means an `available-me` can be generated, but wrappers also catch `UnavailableExceptions` in order to generate `unavailable-mes`. The wrapper methods also update local storage as necessary based on return values from RMI calls. Additionally, each wrapper method also calls the *Network Simulator* subcomponent to simulate how the current network type will affect the call. This component periodically changes to a new random connection type, selecting from dail-up modem, broadband, LAN or WLAN, each of which introduces different network delays and changes the application's connection cost. This change triggers a `cxn-cost-changed-event`, which is global rather than belonging to a specific interaction and is thus published to a topic rather than queue. The connection cost, along with the value of a new file and hosting costs allows newly initialised interactions to be assigned realistic outcome-costs at start up.

6.3 Preliminary Results

The file server scenario has the scope, duration and complexity to permit a variety of interesting experiments to be defined. To give a taste of the kind of experiments, we consider a situation where feedback is desired when the integrity of any file from the application is undermined on a server. In this case, feedback tells the application to terminate all of its interactions with this server. We can express such a rule in the Jess language, using our generic functions as follows:

```
(defrule preliminary-experiment (integrity-undermined-event
  (appID "fileserverapplication")(xnID ?xn)(trusteeID ?server))
  => (notify-xn-app ?xn (str-cat "TERMINATE-ALL:" ?server)))
```

To evaluate the usefulness of this rule, the simulation models a storage fault that gradually propagates through the server affecting more and more files on the way. The server does not notify of any of the integrity breaches and the application must do so when it discovers the problem. The evaluation compares the number of files corrupted on file server when no rule is specified, then runs a trace replay with the rule to determine how many files were saved by early removal, one file at a time. The experiment is run with 100 files at different speeds of corruption propagation, the averages number of corrupted files from 5 runs at each speed seen in figure 4.

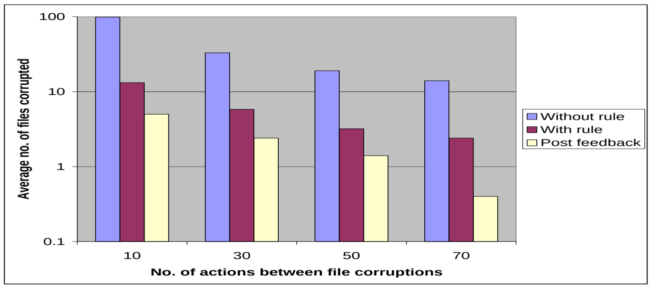

Fig. 4. File corruption experimental results

As can be seen, the number of files saved is is proportional to the frequency of corruption, and in each case around 84% of files were saved, with around 35% of the corruptions occurring after the first termination call.

7 Conclusions and Future Work

We have presented our model of interaction, a monitor architecture and a prototype implementation. Together these support detailed evidence collection during trust-based interactions and can guide a decision process by providing useful, relevant, detailed feedback promptly and in an automated manner. While the model requires the developer to put in more effort to define a particular type of interaction up front, this alleviates the burden on the user at runtime to provide detailed feedback. Our evaluation is still in the preliminary stages, but as seen in section 4, our monitor can provide useful and prompt feedback. We intend to continue validating the monitor using more rules in the file server scenario and also using other scenarios within a more realistic context.

References

1. A. Abdul-Rahman and S. Hailes. Supporting trust in virtual communities. In *Proceedings of the 33rd Hawaii International Conference on System Sciences-Volume 6*. IEEE Computer Society Press, January 2000.
2. Mariano Cilia, Christof Bornhovd, and Alejandro P. Buchmann. Moving active functionality from centralized to open distributed heterogeneous environments. In *CooplS '01: Proceedings of the 9th International Conference on Cooperative Information Systems*, pages 195–210. Springer-Verlag, 2001.
3. Colin English and Sotirios Terzis. Monitoring interactions between trusting entities *A Simulation-based Analysis*. Technical Report 02 (to appear), University of Strathclyde, Computer and Information Sciences, December 2005.
4. Colin English, Sotirios Terzis, and Paddy Nixon. Towards self-protecting ubiquitous systems: monitoring trust-based interactions. *Personal and Ubiquitous Computing*, November 2005.
5. V. Cahill et al. Using Trust for Secure Collaboration in Uncertain Environments. In *Pervasive Computing Magazine*, volume 2, pages 52–61. IEEE Computer Society Press, 2003.
6. Ernest Friedman-Hill. *Jess in Action*. Manning Publications, 2003.
7. The Mandarax Project Homepage. http://mandarax.sourceforge.net/.
8. The Mantaray Project Homepage. http://www.mantamq.org/.
9. The RuleCore System Homepage. http://www.rulecore.com/.
10. Adun Jøsang, Elizabeth Gray, and Michael Kinateder. Analysing topologies of transitive trust. In Theo Dimitrakos and Fabio Martielli, editors, *Proceedings of the Workshop on Formal Aspects of Security and Trust (FAST2003) at FM2003*, volume TR-10/2003 of *IIT Technical Reports*, pages 9–22, Pisa, Italy, September 2003.
11. Mogens Nielsen and Karl Krukow. On the formal modelling of trust in reputation-based systems. In J. Karhumki, H. Maurer, G. Paun, and G. Rozenberg, editors, *Theory is Forever: Essays Dedicated to Arto Salomaa*, volume 3113 of *LNCS*, pages 192–204. Springer, 2004.
12. Norman Paton and Oscar Diaz. Active database systems. *ACM Computing Surveys*, 31:63–103, March 1999.
13. Sotirios Terzis, Waleed Wagealla, Colin English, and Paddy Nixon. Trust lifecycle management in a global computing environment. In C. Priami and P. Quaglia, editors, *Post-Proceedings of the Global Computing 2004 Workshop*, volume 3267 of *LNCS*, Roveretto, Italy, 2004. Springer.
14. Li Xiong and Ling Liu. A reputation-based trust model for peer-to-peer ecommerce communities. In *Proceedings of the 4th ACM conference on Electronic commerce*, pages 228–229, San Diego, CA, USA, 2003. ACM Press.