A Tool for Parallel Distributed Reasoning about Computing Networks

S. Musman
MITRE Corp
7515 Colshire Drv
McLean, Va, 22153
smusman@mitre.org

F. Duncan
Augmented Systems, Inc
7400 Seabrook Lane
Springfield, Va, 22153
fduncan@aug-sys.com

Abstract

We describe a distributed reasoning system called Otto-Mate. Otto-Mate provides a variety of built-in agents that make it easy to ingest events from logs and add-on tools, response agents to implement responses, and reasoner agents that can reason about situations. All of these capabilities are connected using a popular agent framework, enhanced with a few additional security features. Reasoning agents in Otto-Mate implement a form of situational reasoning, related to case-based reasoning, and the facts in a reasoner’s working memory can be synchronized over multiple reasoners. This allows the implementation of parallel distributed reasoning algorithms that can detect event patterns irrespective of whether the events occur locally or remotely. Distributing the reasoning avoids a single point of failure for the monitoring system, and makes the system much more robust and survivable.

Overview

Reasoning about computer and network events remains the cornerstone of automating management, security and health monitoring problems, but reasoning in this domain is complicated. Events and reasoning occur at different levels of granularity and is further complicated by the lack of standards for events that are reported. Operating, network components (i.e. switches), and add-on tools (i.e. IDS, logs, virus scanner) all report differently, yet each monitored resource often provides evidence that yields part of an overall picture once the events are correlated.

Despite the maturing of agent-based systems a majority of commercial monitoring tools remain centralized. Centralized approaches introduce a single point of failure for the system, leaving the monitored components unprotected if they can be separated from the central monitor. The centralized approach is ultimately not scalable.

Systems such as AAFID [Spafford 00] were early attempts, to apply distributed reasoning to this problem domain. Implemented in Perl, it had performance issues, and AAFID contained none of the necessary management, security or performance features to function as an operational system. Many other research systems are similar and are not useful for practical deployment.

Expert systems such as P-BEST [Lindquist, 99], have been applied to reasoning about security events. Such systems are “event” driven, allowing rules to be defined to detect events, or simple event patterns. These systems usually have no explicit knowledge of time, and as such cannot easily reason about events that are expected to occur at certain times, or detect missing events. Historically, most of the knowledge they encode represents shallow knowledge of the domain.

To better understand the reasoning requirements, we describe a sample problem. An earlier version of the Otto-Mate reasoner [Musman, 00] was used in a DARPA sponsored experiment in Autonomic Information Assurance [Theriault 02]. A reasoner agent was connected into Survivable Autonomic Response Architecture (SARA) [Lewendowski 01], where the task of the reasoner agent in the experiment was to defend a computing network against

Figure 1: Reasoning States for email Virus Defense
DoS attacks caused by 0-day email viruses. A variety of sensors and some novel algorithms were developed for the task. Email client sensors, could detect suspicious email activity such as generating email messages faster than a human can possibly type, or sending email messages immediately after opening an email or its’ attachments. Email activity could also be linked together into email flow trees that can act as a detector because the propagation pattern of email viruses is often quite uniquely different from legitimate traffic. An email queue size detector [Gupta, 03], monitored the overall state of the email server and detected if the system became bogged down. Response mechanisms in this system could purge email messages labeled as containing a virus, and also dynamically insert message-blocking signatures at the email server that match certain criteria. A state diagram for the system is shown in figure 1.

This example illustrates several aspects of reasoning about computing events. It demonstrates a need to:

- **Assess system state:** By analyzing and correlating events to determine if the system is under attack, or noting a response is ineffective and that something else should be done about it;
- **Select the right response given the system state:** By recognizing a certain class of attack and the type of response that could resolve the problem. Responses can also involve starting sensors looking for certain virus symptoms, dynamically adapting system state (sensing in this instance);
- **Assess alert relevance given system state:** For a non-polymorphic virus response, if a blocking signature is in place all future alerts should be associated with messages that were delivered before the blocking operation. Alerts from messages created after the blocking is applied indicate that there a problem remains;
- **Ensure responses are timely:** Ensure that responses are carried out, request them again if necessary, select an alternative or alert a human.

The SARA architecture supported only one reasoning agent at a time, acting as a centralized reasoning solution to the problem. As a result, on some occasions the network saturated, and severe SARA message latencies (i.e. in excess of 5 minutes wall clock time) occurred and contributed to test scenarios in which the virus succeeded.

Often event patterns and relevant responses occur at the level of individual components (i.e. most activity on the email server relates only to its operation). Patterns of events that reflect larger system level patterns are less common, so a distributed reasoning solution that did not centralize the processing would clearly be more efficient and was a motivation for the design of our system. Our analysis, illustrates that a distributed approach could reduce events distributed over the network by over 65%.

**System Description**

Otto-Mate consists of a variety of different agents classes: resource monitoring agents, reasoning agents, and response agents. Resource monitoring agents monitor assigned resources and pass events into the Otto-Mate system. Reasoning agents include those described below, but also include anomaly detection agents [Varaandi, 03]. Response agents implement the resource specific actions to resolve problems. Though the base functions of these agents are simple, most of the agents also have additional capabilities to support drill-down.

Our reasoning model allows for situations that include a combination of known bad situations, policy violations, as well as anomaly models of unusual events.

Because computing networks are usually heterogeneous, event normalization is required to share information. Despite efforts such as IDMEF [IETF 04] this remains an area in which more work needs to be done. Efforts such as IDMEF provide more guidance on how to say things, rather than on what should be said. As such, they do not provide a standard ontology of the domain.

---

Figure 2: Normalizing Events and Requests
Our knowledge-base structure is illustrated in figure 2 where most of the reasoning occurs in the middle layers, and relates to generic abstracted events. This significantly reduces the need duplicate effort developing situations for each separate resource type (i.e. Unix vs. Windows). Other differences (i.e. what might be an unusually high CPU load on a workstation would be different on a server) are addressed by being able to assign the agents to groups, representing both is-a and has-a relationships. Situations and reasoner variable values can be assigned to the groups and the intent is to be able to reason about resource types, based on location or function. Configuring every resource individually would be unscalable.

Reasoner agents use facts to represent information (though facts can contain confidence values). Events are represented as inheritable knowledge structures with named fields. Facts can be ordinary facts, timer-facts, or event facts. Event facts are transitory, and are automatically removed from the system if no situations match them. Timer facts are special facts associated with a timer mechanism built into the reasoners. Timer facts can be defined to exist for specific periods of time, or be used as timed (sliding window) counters so that they can detect event frequencies given a count and timeframe. Timer facts can also be scheduled for assertion at a future time, delaying when the reasoner operates on them.

Facts and timers also have fields called; “persistent” and “group”. Setting the persistent value causes the reasoners to cache the facts on disk until they are retracted, so they will continue to exist even if the reasoners are stopped and restarted. The group field can be used to specify that a fact is synchronized across multiple reasoners. When assigned a specific group, the fact becomes visible to all reasoners running on resources that are part of that group. Changes to this fact, occurring on any of the reasoners, will be seen in all of the reasoners in the group.

An example of detecting more than 5 failed login attempts in 5 minutes is shown in figure 3. This type of brute-force password guessing that is still surprisingly common. As the timer fact “failed-access-from-user-counter” belongs to the “DMZ” group if the remote-login-failures occur across any one or more of the computers in that group the value is incremented on all of them, turning this into a parallel distributed algorithm.

Although not illustrated, other situations were programmed to take care of responses: blocking the host for a short period of time, re-enabling it later. By issuing only a temporary blocking of the host, most brute force attacks are repelled without creating any long term denial of service situation, and there is no need to keep track of large blocking lists. Additional situations could also monitor additional activity from hosts that had incidents, and can escalate the severity of response for repeat offenses.

In Otto-Mate situations can exactly match (like a rule), or partially match. Situations have a default priority, but this priority can also be set to define the order in which matching situations are used when more than one matches. Situations also have function switches call-next, and always-fire that allow them to layer functionality. Call-next implements the response actions of the best-matching situation, and then executes a matching situation with the next highest priority (if any). The always-fire switch causes a matching situation to fire irrespective of whether or not there are any other matching situations. The general consensus when programming rule-based systems is that forcing rules to execute in a specific order is poor form. In Otto-Mate we violate that principle to reuse existing situations, or to supersede behavior of certain situations.

Sometimes reasoner agents request additional information from other agents. The decision of where to query for information is either determined statically, based on system layout, or dynamically from location information in events (i.e. an ip address or host name). The information returned can be current values, or historic information and the reasoner is able to reason about both simultaneously. An example of a useful information query might be a reported access failure from an internal host to a sensitive database. Here there would be a desire to determine who was logged
into the unauthorized host accessing the database, and determine who made the access attempt.

Queries for information, can be returned to, and processed by the requesting agent. Alternatively the requester can delegate the decision making to another reasoning agent using the "reply-to" feature for the request, to tell the owner of the requested information where to send it. This ability to delegate decision making is particularly valuable if there are multiple hypotheses to explore.

**Practical Considerations**

Otto-Mate compares to a variety of other systems for which there are a variety of common methods: rule-based; model-based; fault-trees; CBR; Bayesian Networks; and others [Steinder 01]. Otto-Mate supports CBR, anomaly, and rule-based approaches, and some limited work has been done allowing Bayesian networks to reason about facts in the working memory of the reasoners.

Otto-Mate functions in addition to the other the tasks that the components it monitors perform. Under a typical load, of 30 agents monitoring an average host (processes, system logs, file-access, etc.), the load created by Otto-Mate occasionally spikes at 10% (events are often bursty), but usually is much less than 1%. The reasoner itself can process over 5000 events/sec and the agent framework used is JADE, which utilizes several layers of messaging to improve efficiency [Vitaglione 02] can send 1000’s of messages/sec.

**Summary**

Otto-Mate was developed to automate monitoring, analyzing, correlating, and response to events found in computing systems. In contrast to most commercial systems that do central monitoring, Otto-Mate, reasoning agents are distributed throughout the computing network so there is no single point of failure for the system.

Reasoner agents can also work collaboratively and can query event monitoring agents for current, or historical events that may confirm or deny a current hypothesis. Additionally, reasoning agents can be tightly coupled together. Facts within their working memories can be synchronized across multiple reasoner agents allowing multiple reasoners to reason about related events occurring locally or remotely. In essence multiple reasoners work together to implement distributed parallel reasoning algorithms. A limited version of Otto-Mate is available for download and experimentation from http://otto-mate.aug-sys.com.

**References**


