An AI-ESTATE Conformant Interface for Net-Centric Diagnostic and Prognostic Reasoning

Houston King, Nathan Fortier, and John W. Sheppard
Department of Computer Science
Montana State University
Bozeman, MT 59717
houston.king@msu.montana.edu, nathan.fortier@cs.montana.edu, john.sheppard@cs.montana.edu

***Abstract*—IEEE Std 1232-2010 Standard for Artificial Intelligence Exchange and Service Tie to All Test Environments (AI-ESTATE) provides a standardized approach to modeling and control of a data-driven diagnostic and prognostic reasoning environment. This work continues a previous effort to build wider acceptance and extension of this standard through development of the SAPPHIRE$^{\mathrm{TM}}$ tool. Previously, SAPPHIRE$^{\mathrm{TM}}$ was extended to contain a complete implementation of all required Reasoner Manipulation Services listed by the AI-ESTATE standard. In this paper, we report the results of abstracting this services API, allowing the tool to be run as distinct client and server processes that may communicate over a networked connection. We describe the architecture underlying this client-server interface and the XML protocol constructed to facilitate communication between the client and server.**

I. Introduction

The IEEE Std 1232-2010 Standard for Artificial Intelligence Exchange and Service Tie to All Test Environments (AI-ESTATE) standard [1] provides platform-agnostic models and a service interface facilitating the exchange of diagnostic knowledge and reasoning capabilities. The Numerical Intelligent Systems Laboratory at Montana State University has been working continuously on promoting and enhancing this standard. Within AI-ESTATE is defined a set of Reasoner Manipulation Services (i.e., a service API). These services provide a standardized way to interact with a reasoning server for the purposes of diagnostic and prognostics. Previously, the AI-ESTATE service API had been implemented locally within our tool, called the Standards-based Analysis Platform for Predictive Health and Intelligent Reasoning Environment (SAPPHIRE$^{\mathrm{TM}}$).

Within this document, we describe a net-centric architecture that abstracts the Reasoner Manipulation Services provided by AI-ESTATE to the network level. The client-server architecture is described in detail, as well as the XML-based messaging system that facilitates the communication between a reasoning client performing a diagnostic session and a reasoner server hosting the data-driven inference engine. Identified with this abstraction are a variety of immediate advantages. The server and client can be moved to distinct physical locations and allowed to communicate via a network connection in a standardized way. The protocol derived from the AI-ESTATE services allows third-party clients and servers to communicate with our tools directly. Data-driven diagnostics and prognostics can be computationally intensive: centralizing the task of inference to a particular server can minimize the cost of adding clients to the network.

II. Background

A. AI-ESTATE

Project Authorization Request (PAR) 1232 was approved in February 1990, authorizing the IEEE Standards Coordinating Committee 20 to begin development of the AI-ESTATE standard [2]. The standard was proposed as a way to incorporate artificial intelligence (AI) into testing and diagnostics applications. Since then, the standard has continued to grow and mature—most notably, the standard includes methods for platform-agnostic diagnostic model exchange, as well as a standard application programming interface (API) for interacting with a reasoning engine. The most recent iteration of the standard, IEEE Std 1232-2010 [1] schematically defines four different models using the EXPRESS data modeling language language: fault trees, D-matrices, logic models, and Bayesian networks. Derived from these EXPRESS models were eXtensible Markup Language (XML) schemata, allowing the diagnostic models and session information to be encapsulated and validated via XML documents [3].

An early version of AI-ESTATE was evaluated over several phases of a U.S. Air Force funded SBIR to demonstrate the feasibility of implementing the standard in a component-based Automatic Test System (ATS) [4]. Under support from the U.S. Navy, prototype tools were developed and a controlled experiment run to demonstrate the model exchange [5] and service interoperability [6] of the standard across reasoners. A previous iteration of our current project introduced an extension to the Bayesian Network Model (BNM) and services API [7]. This extension permits the SAPPHIRE tool to model and interact with Dynamic Bayesian Networks (DBNs). Before this extension, the AI-ESTATE standard had only been used to encapsulate and infer diagnostic data. With this work, the standard became capable of modeling and inferring prognostic data represented through a DBN.

B. Bayesian Networks

A Bayesian network is a probabilistic graphical model that represents a joint probability distribution over a set of random variables as a directed acyclic graph [8]. Each node in a Bayesian network represents a random variable, while edges between nodes represent probabilistic relationships between the variables. For any set of random variables in the network, the probability of any entry of the joint distribution can be

computed as:

$$P(\mathbf{X}) = \prod_{i=1}^{n} P(x_i|\mathbf{Pa}(x_i))$$

where$P(\mathbf{X})$ is a joint probability distribution over the variables $x_1, ..., x_n \in \mathbf{X}$ in the network.

The Bayesian network model (BNM) defined by AI-ESTATE is used to represent a specific type of Bayesian network used for diagnostics [1]. In the BNM, nodes are represented using BayesDiagnosis (which can be subclassed further into BayesFault and BayesFailure) and BayesTest entities. BayesDiagnosis entities are not dependent on any other entities while BayesTest entities can be dependent on both BayesDiagnoses and other BayesTests.

C. Dynamic Bayesian Networks

A Dynamic Bayesian Network (DBN) extends the Bayesian network by relating variables to one another over discrete time slices [8]. Each time slice contains a traditional Bayesian network indexed by time t. The conditional probability distribution of a variable at a given time-slice can be dependent on itself or other variables over any number of previous time slices. Variables in first-order DBNs are only conditionally dependent on variables within the preceding time slice. Thus, the full joint distribution for a first-order DBN can be computed as:

$$P(\mathbf{X}_0, ..., \mathbf{X}_k) = P(\mathbf{X}_0) \prod_{t=1}^{k} P(\mathbf{X}_{t+1}|\mathbf{X}_t)$$

A DBN can utilize evidence about previous time slices to reason about the state of variables in future time slices. First-order DBNs are often defined by specifying a prior network $\mathbf{X}_0$ and a temporal network $\mathbf{X}_t$. DBNs are commonly applied in areas that require inferences based on sequences of observations over time. While DBNs are often used to perform inference on the current time slice based on prior observations, DBNs can also be unrolled further to forecast future states based on the conditional probabilities defined in the network. Thus, when performing inference at some time slice k, the temporal network is typically unrolled into $\mathbf{X}_1, \mathbf{X}_2, ..., \mathbf{X}_k$.

Within SAPPHIRE$^{\mathrm{TM}}$, we have extended the AI-ESTATE BNM to support DBNs for use in diagnostics and prognostics. These extensions include the introduction of temporal links from diagnosis-to-diagnosis, test-to-test, and diagnosis-to-test. Unlike previous links between DependentElements, these temporal links allow for self-referencing elements.

D. Reasoner Manipulation Services

The AI-ESTATE reasoner manipulation services define procedures and functions used to interact with a diagnostic reasoner application during a diagnostic session [1]. The functions defined by the services allow the reasoner to: recommend actions to the client, perform inference over a model, receive test information from the client, set checkpoints for session recovery, backtrack to undo service calls, and provide the client with diagnostic conclusions. The reasoner manipulation services are independent of any one type of model and, as a result, the specific implementation of the services may vary.

Within SAPPHIRE$^{\mathrm{TM}}$, we have implemented the AI-ESTATE API consisting of all required reasoner manipulation services specified by the standard. This API makes use of the Dynamic Context Model (DCM) and Reasoner Service Model (RSM) entities to allow the functions and procedures defined by AI-ESTATE to be executed on a variety of different systems. The functions defined in the API provide feedback to the client and make changes to DCM entities as specified in AI-ESTATE. The services API makes use of several inference algorithms to infer diagnostic conclusions and recommend potential actions to the client.

III. Architecture

The major design goal of the work reported in this paper is the division of the SAPPHIRE$^{\mathrm{TM}}$ tool into distinct, network-capable processes. The reasoner client process is designed to be lightweight; the data-driven reasoner and inference engine are never used on the client when the tool connects to a remote reasoner server. The reasoner server is a headless process controlling threads to which reasoner clients may connect to start a reasoning session. Common to the client and server is a small communication stack that manages the transmission of messages between the client and server.

A. Communication Architecture

The reasoner client and server share several classes that facilitate the construction and sharing of XML messages over a networked interface, namely the ReasonerXMLBuilder, ReasonerXMLReciever, and ReasonerXMLSender classes.

The ReasonerXMLBuilder class handles the majority of the workload for constructing and parsing the XML messages that are transmitted over the networked connection between the client and server. For example, the *initializeDiagnosticProcess* service, responsible for initializing a new reasoning session, requires two strings as arguments: an *itemID*, which is the unique identifier for the system being diagnosed or prognosed, and a *systemItemName*, which gives the name of the system item under test. To facilitate the construction of such a message, three calls are made to an instance of the ReasonerXMLBuilder class. The first call is to a function that creates a template `function-call` document, that contains various parameters required for validation, as well as the name of the service the message is intended to call. The next two calls are to a helper function that takes the template `function-call` document and adds string elements with the associated IDs and values. Example results of such calls can be seen in Figure 1.

Upon receipt of a message by the reasoner server thread, the server's copy of ReasonerXMLBuilder is used to parse the needed information from the XML document. The class parses the service name from the document, then an ancillary function uses the parameter IDs associated with the service to extract the requisite parameters. In the case of Figure 1, the strings for *systemItemName* and *itemID* would be extracted from the document. After extracting all parameters, the server invokes the named service. After the service is completed, ReasonerXMLBuilder is used to construct a `function-response` message and to attach any return parameters. The document is sent back to the client, and the process is then allowed to repeat.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<function-call
xmlns:bnm="urn:oid:1.3.111.2.1232.100.2011.2"
xmlns:exp="urn:iso:std:iso:10303:-28:ed-2:tech:XMLschema:common"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:oid:1.3.111.2.1232.100.2011.2 Ai_estate_bnm.xsd">
<fname>initializeDiagnosticProcess</fname>
      <params>
            <string id="itemID">BayesNetworkModel-7211aa37-153e-4dab-aace-
            7c4cde511c58</string>
            <string id="systemItemName">Doorbell System</string>
      </params>
</function-call>
```

Fig. 1: Example *initializeDiagnosticProcess* XML Document

The ReasonerXMLSender and ReasonerXMLReceiver classes handle the actual transmission and receipt of the XML messages between the client and server. Both classes facilitate communication by managing a socketed connection to the designated client or server. The ReasonerXMLSender begins the communication process by accepting an XML document to be sent over the connection. After the document is passed to the class, the sender validates the document against a variety of XML schemata, including schemata for XML-based representations of objects from the AI-ESTATE standard. This validation is required, since many parameters passed via the service API are objects drawn directly from the Bayesian Network Model and Common Element Model of AI-ESTATE.

Once a message has been validated, whitespace is stripped, and the message length in characters is calculated. The sender transmits the length in characters over the socket first, notifying the receiver of the size of the incoming message. Following this, the message itself is transmitted over the connection. The ReasonerXMLReceiver first receives the number of characters in the incoming message, then reconstructs the message from the expected number of incoming characters. After this reconstruction, the message is again validated, and after validation the message is handed up to the reasoner server for processing.

B. Reasoner Client

The reasoner client utilizes the same service API as that used in the previous iteration of SAPPHIRE^{TM}. From this abstraction, the transition between a local reasoning session (without a networked reasoning server) and a remote reasoning session (with a networked reasoning server) is seamless. In this sense, the networked reasoner client is simply a surrogate for the local reasoner—the GUI for the tool is identical in both instances. Additionally, while SAPPHIRE^{TM} provides a reference implementation for the service API and for the network interface described in this document, the XML-based abstraction between the server and client permits other tools to be used interchangeably with this client and server. Provided a client or server replied using the same XML formatting as that put forth in this work and as in the AI-ESTATE standard, intercommunication between clients and servers should be trivial.

C. Reasoner Server

The reasoner server implements two primary classes to provide its intended function: ReasonerServer and ReasonerWorker. ReasonerServer acts as the driving class, handling the establishment of a socketed connection and dispatching ReasonerWorkers to handle reasoning sessions with connected clients. The server itself is designed using the fixed-size thread pool paradigm: a fixed, specified, number of threads (ReasonerWorkers) are initialized when the server is first started. Each of these threads is dedicated to a client upon connection, and, if all threads are allocated to a client, then the server will block until one of the existing sessions ends. Motivating the use of the fixed-size thread pool is the computational complexity of data-driven inference. Though notable gains have been made in reducing this cost of inference (see [8]), it remains an expensive process. The thread pool allows an administrator to control how many simultaneous clients are appropriate for a given server.

Each ReasonerWorker thread handles direct input from the connected client. Once a session is initialized, the worker loops, waiting for a `function-call` document from the client. Upon receipt, the function name of the service is parsed from this document and any additional parameters for the given service are pulled from the document. The worker then invokes its local reasoning engine, performing the desired service. After completion, the worker builds and transmits a `function-response` message to the client. Upon receipt of a designated `close-connection` message, the worker stops its main loop, closes the connection to the client, and returns the thread to the thread pool for later use.

IV. XML-Based Protocol

The eXtensible Markup Language (XML) is a widely-used human-readable markup language that provides platform agnostic encoding of information. In previous work [7], the SAPPHIRE^{TM} tool had been developed to create, process, and validate objects in XML that were defined in accordance with the AI-ESTATE standard. The XML-based representations of objects from the Common Element Model (CEM), Bayesian Network Model (BNM), and other parts of the AI-ESTATE standard were found to be effective for encapsulating the information required for the reasoning process. This efficacy

```
<double id="minConfidence">
      0.05
</double>
<integer id="timeStep">
      14
</integer>
```

Fig. 2: Example Primitive XML Objects

even held when the models were extended to represent Dynamic Bayesian Networks, as described in [7]. These previous successes, the human-readability of XML documents, and the pre-existing XML-centric tools in SAPPHIRE, motivated the decision to use XML as the basis for communicating over a networked interface, even when such messages may suffer from larger size.

A. Protocol Overview

The protocol used for this project is straightforward, and in many ways, is a natural and network-focused extension of the Reasoner Manipulation Services defined within AI-ESTATE. A normal reasoning session is expected to proceed as follows:

1) A client establishes a socket-based connection with a network-based host known to be running an instance of the ReasonerServer. Upon connection, the server dispatches one of the reasoning server threads, which will remain dedicated to the current client until the completion of the reasoning session.
2) For each service evocation, a `function-call` XML message is generated and sent from the client to the server, commanding the server to run the desired service. After completing the service, the server replies to the client by constructing and sending a `function-response` XML message corresponding the completed function call. If the service defines any returned parameters, or if any errors are encountered in completing the service, they will be encapsulated within this `function-response` message.
3) Once the reasoning client has processed all desired services, a `close-connection` message is sent to the server. Once this message is received, the reasoner's thread shuts down and the socket between the client and server is severed.

All messages transmitted over the socket between the client and server in this process are proceeded with an integer indicating the number of characters in the following message.

B. Primitive Objects

The primitive objects in the protocol are represented by fully-contained XML nodes within a `function-call` or `function-call` message. These structures are those that have a direct correspondence with the basic structures in most programming languages, such as strings, integers, and enumerations. Figure 2 shows examples of primitive objects that could be contained within a message.

C. AI-ESTATE Objects

Many of the service calls within the API involve passing or returning AI-ESTATE objects. From [7] there is a pre-existing framework for marshaling and unmarshalling XML representations of all AI-ESTATE objects used in the service API. This marshalling and unmarshalling works with an object-reference system to avoid repetition of potentially large XML objects. This object-reference system has been incorporated into the protocol. Any transferred AI-ESTATE objects are split into a reference and the object itself. The references are a child in the `params` or `results` nodes of `function-call` and `function-response` messages, respectively. The objects themselves are contained within a separate, dedicated `objects` child of the message. All of these objects are validated upon transmission and receipt by the client and server. Figure 3 shows a snippet from a `function-call` message, showing the `params` and `objects` nodes. The `params` node only contains the reference to the object, whereas the `objects` node contains the complete object.

D. Function Call Messages

Function call messages are sent from the client to the server when the client wishes to evoke a particular service from the API. Each `function-call` message is a self-contained XML document describing the service to be evoked as well as all parameters required for the function call. All `function-call` messages can have up to three child nodes: an `fname` node, a `params` node, and an `objects` node. The value of the `fname` node is simply the name of the service to be called by the server. The `params` node contains the parameters required for the function call. If a parameter is a primitive object, it is fully contained within the `params` node. If a parameter is an AI-ESTATE object, the node will only contain a reference to the object. The `objects` node contains the full XML representations of AI-ESTATE objects passed for a service call. The `fname` node is always required within a `function-call` document, whereas the `params` and `objects` nodes can be omitted if there are no parameters or AI-ESTATE objects, respectively.

Described in [7] are extensions to the AI-ESTATE standard permitting prognostics using Dynamic Bayesian Networks (DBNs). Included in this previous work was the addition of a *timeStep* parameter to several services in order to specify the step to which the service applies. For example, the *getDiagnosticResults* service calculates a list of diagnostic conclusions using the inference engine. With the prognostic capabilities of a DBN, these diagnostic conclusions could be calculated for any arbitrary future timestep, so specifying the time at which the service applies is required. The XML protocol handles this difference between models in a straightforward manner: if a service is to be applied at or relative to a particular timestep, a `timeStep` element is added to the `params` node present in every `function-call` message. If the model does not require a specific timestep, then this element is omitted from the outgoing `function-call` message.

E. Function Response Messages

Function response messages are sent from the server to the client after completion of a service invocation. The structure of

```
<params>
      <actions>
            <dcm:Actualaction ref="ActualAction-1657888f-0fd0-4825-8c76-
            9b1257697b93" xsi:nil="true"/>
      </actions>
</params>
<objects>
      <dcm:Actualaction id="ActualAction-1657888f-0fd0-4825-8c76-
      9b1257697b93">
            <Actionname>push</Actionname>
            <Statusvalue>FAIL</Statusvalue>
            <Statusconfidence>1.0</Statusconfidence>
            <Timeperformed>0</Timeperformed>
      </dcm:Actualaction>
</objects>
```

Fig. 3: Example AI-ESTATE object parameter within an XML message.

a `function-response` message strongly mirrors that of a `function-call` message. The message contains (up to) three child nodes: an `fname` node, a `results` node, and an `objects` node. The `fname` node names the service call being responded to, the `results` node contains any returned primitive objects or references to AI-ESTATE objects, and the `objects` node contains the full XML representations of any returned AI-ESTATE objects.

F. Error Handling

The AI-ESTATE standard defines a variety of errors that can occur in the course of interacting with a reasoner. To facilitate reporting these errors in a standardized way, there are a variety of status codes described as part of the Reasoner Manipulation Services. Originally, SAPPHIRE^{TM} implemented these various status codes as exceptions that may be thrown in the course of processing a service. With the extension of the tool to the networked interface, these status codes were incorporated into the `function-response` messages. Included in the `function-response` node is a `statusCode` attribute. This attribute will have a value of `OPERATION_COMPLETED_SUCCESSFULLY` if the service was able to complete as intended. In the event of a failed service call, the attribute will instead equal the status code corresponding to the failure, such as `INVALID_MODEL_SCHEMA` or `MISSING_OR_INVALID_ARGUMENT`. If a client attempts to call a service that does not exist, the server will return a `SERVICE_NOT_AVAILABLE` error.

V. Example Transmission

Figure 4 demonstrates a full end-to-end service call between a reasoner client and server. First, the client calls *initializeDiagnosticProcess* locally, passing the *itemID* and *systemItemName* as required by the standard. Then, using the ReasonerXMLBuilder class, a `function-call` document is created for transmission. This document contains the name of the function and the parameters required to call the service. The document is sent from the client's ReasonerXMLSender to the server's ReasonerXMLReceiver. The ReasonerWorker thread responsible for the session between this client takes the document and passes it off for processing locally. After the service completes, the worker generates a `function-response` document to inform the client of the successful call. Included in this document is the *sessionName* string, which, as per AI-ESTATE, is the unique ID of the current session. This document is sent from the server to the client, allowing the process to continue.

VI. Information Assurance and Security

Development of our client and server tools has borne a variety of advantages. Formerly, the tool was a single process, and any machine running the tool was required to run all tasks simultaneously. With the client and server architecture, this is no longer the case: separate physical machines can run the distinct processes. Beyond this, hardware is customizable to these processes: clients are lightweight and require few resources, while the server can be made robust to deal with the computational cost of inference. Various data files can be centralized to the server, including reasoning session information, reasoning models, test results, and maintenance information. However, even with these advantages, we strove to remain cognizant of the security and information assurance implications of these architecture changes. With the Department of Defense funding this work, the necessity of risk mitigation has been kept at the forefront through our development. As such, we have considered a variety of areas where we can protect the data used by the client and server in the pursuit of developing a robust and secure tool.

The first area we consider is control of user access. With a centralization of the server and data repository, it would be possible to add a permissions layer to the tool. This permissions layer could control which clients can access which models, increasing control over the distribution of model information and reducing the risk associated with having these models available and accessible in a networked setting.

Second, as part of the efforts towards standardization, we have developed the tool to use IEEE Std 1636.1 Test Results and Session Information [9] and IEEE Std 1636.2 Maintenance Action Information [10] files. As indicated by the security classification attributes associated with entities defined by these standards, conformant files may be sensitive and have policy restrictions, thus disallowing ubiquitous access. Our client/server architecture would again be amenable to including

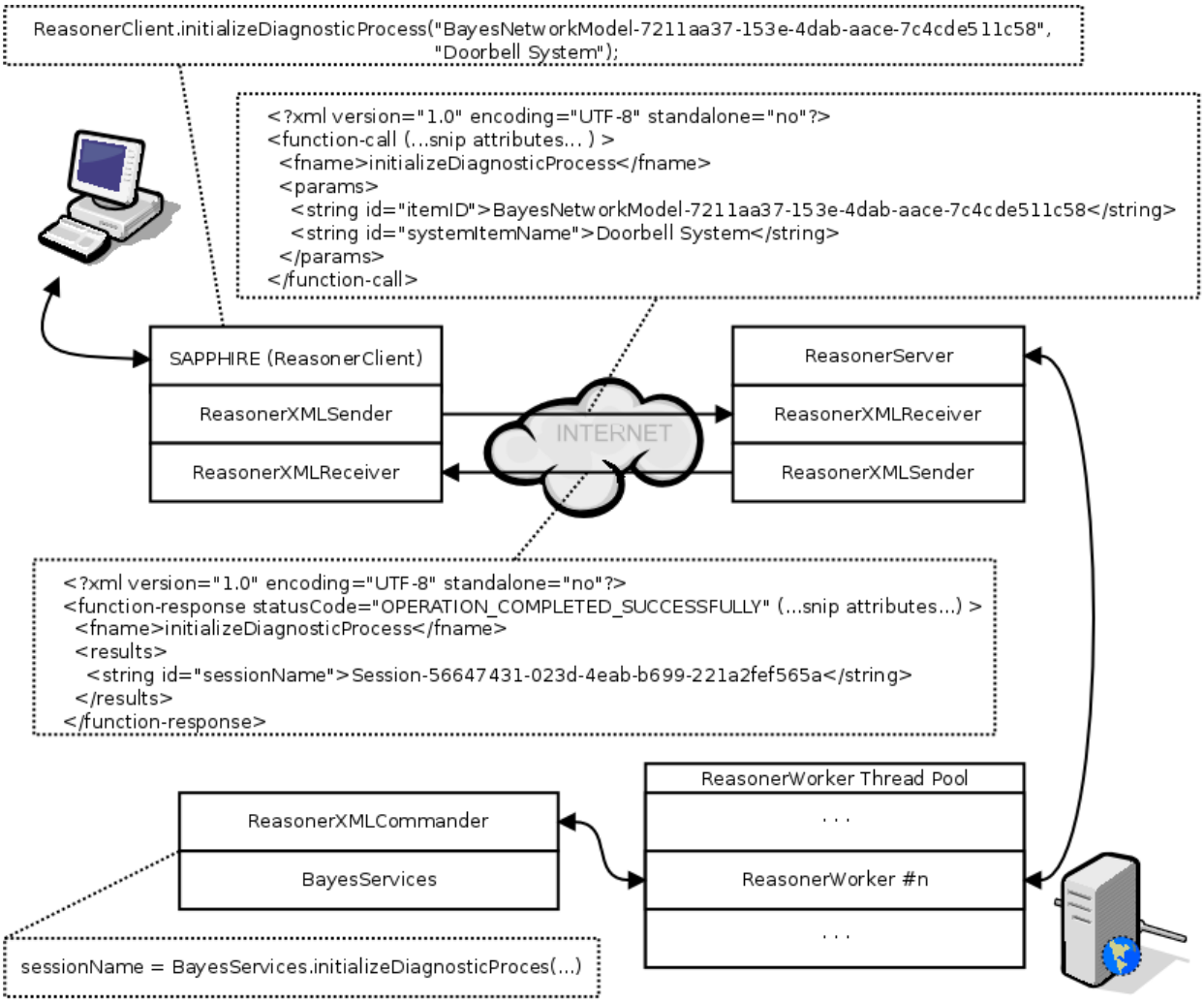

Fig. 4: Example end-to-end communication between reasoner client and server.

user and file permissions that would restrict using these files to only qualified and appropriately cleared parties.

Third, while the XML messages transmitted by the tool are currently plaintext, this need not be the case in general. The ReasonerXMLSender and ReasonerXMLReceiver classes could be extended to permit building an encryption scheme directly into the tool. Alternately, since the tool uses an IP socket based approach, many off-the-shelf solutions such as Virtual Private Networks could immediately secure the transmissions between clients and servers, greatly reducing the risk of interception.

Finally, centralization of data files and reasoning models lends itself to many of the same techniques that have been found favorable for client-server architectures as a whole. A central data repository would simplify preparations for disaster recovery. The data files could be stored using a distributed parity scheme, such as RAID 5, allowing for recovery in the event of drive failure. Furthermore, the data in the repository could be mirrored securely to an off-site location in the event of site failure. The reasoning server process itself could be setup using well-established rollover techniques, allowing for little to no downtime in the event of machine or site disruption.

VII. Future Work

While there are many benefits to this system, it must be noted that this is a preliminary implementation of the networked interface. The protocol itself is fairly naïve XML, which could suffer from a certain degree of bloat: a compression or encoding system may be appropriate to shrink transmission sizes. The messages are currently transmitted in plaintext across the network interface. While this is sufficient for the initial implementation, a certain amount of security or encryption may be desirable when approaching information assurance issues. The reasoning server currently handles session and model information in a naïve manner as well. The system as a whole could benefit from the design and implementation of a centralized data repository to which the server and clients may refer, eliminating the need for transmission of "heavier" AI-ESTATE objects, such as complete Bayesian Network Models. Finally, there has not been a formal study examining the efficacy of the networked interface. Such a study would likely lead towards other significant areas for future improvement.

VIII. Conclusion

In this paper we described a network-focused extension to the AI-ESTATE standard. SAPPHIRE^{TM} has been segmented into two distinct processes: A reasoner client and a reasoner server. These processes communicate via a socketed interface, permitting transmission across a network connection. XML messages corresponding to AI-ESTATE API calls and responses are constructed and sent between the client and server.

Separating SAPPHIRE^{TM} into these distinct processes has yielded a variety of immediate advantages. SAPPHIRE^{TM} now has the capacity to connect multiple reasoner clients from multiple, distinct, remote locations. The reasoner server can be specially constructed to handle the computationally intense reasoning process, with dedicated processing power and memory. This enables the clients to be constructed conservatively, reducing the cost of adding another reasoning client to the system as a whole. Beyond this, the information from reasoning sessions is centralized to the server, simplifying the task of storing the results of the sessions, and potentially eliminating redundancy between storage of reasoning models.

Acknowledgment

This project was supported as an STTR under US Navy contract N68335-11-C-0506. We thank Mike Malesich and Jennifer Fetherman for their continued support. We also thank Patrick Kalgren, John Gorton, Brian Drost, and Myra Torres of Impact Technologies/Sikorsky Innovations for their continued collaboration with this STTR.

References

[1] IEEE Std. 1232-2010, *IEEE Standard for Artificial Intelligence Exchange and Service Tie to All Test Environments (AI-ESTATE)*, Piscataway, NJ: IEEE Standards Association Press, 2010.

[2] L. A. Orlidge, "An overview of ieee p1232 ai-estate. the standard for intelligent reasoning based systems test and diagnosis arrives," in *AUTOTESTCON'96, Test Technology and Commercialization. Conference Record*. IEEE, 1996, pp. 61–67.

[3] ISO 10303-28:2007, *Industrial Automation Systems-Product Data Representation and Exchange-Part 28: XML Representation of EXPRESS Schemas and Data Using XML Schemas*, Geneva, Switzerland: The International Organization for Standardization, 1994.

[4] J. W. Sheppard and A. J. Giarla, "Information-based standards and diagnostic component technology," in *AUTOTESTCON Proceedings, 2000 IEEE*. IEEE, 2000, pp. 425–433.

[5] J. W. Sheppard, S. G. Butcher, P. J. Donnelly, and B. R. Mitchell, "Demonstrating semantic interoperability of diagnostic models via ai-estate," in *Aerospace conference, 2009 IEEE*. IEEE, 2009, pp. 1–13.

[6] J. W. Sheppard, S. G. Butcher, and P. J. Donnelly, "Demonstrating semantic interoperability of diagnostic reasoners via ai-estate," in *Aerospace Conference, 2010 IEEE*. IEEE, 2010, pp. 1–10.

[7] L. Sturlaugson, N. Fortier, P. Donnelly, and J. W. Sheppard, "Implementing ai-estate with prognostic extensions in java," in *AUTOTESTCON, 2013 IEEE*. IEEE, 2013, pp. 1–8.

[8] D. Koller and N. Friedman, *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[9] IEEE Std. 1636.1-2013, *Software Interface for Maintenance Information Collection and Analysis (SIMICA): Exchanging Test Results and Session Information via the eXtensible Markup Language (XML)*, Piscataway, NJ: IEEE Standards Association Press, 2013.

[10] IEEE Std. 1636.2-2010, *Software Interface for Maintenance Information Collection and Analysis (SIMICA): Exchanging Maintenance Action Information via the Extensible Markup Language (XML)*, Piscataway, NJ: IEEE Standards Association Press, 2010.