Adaptive Mirroring of System of Systems Architectures

Nathan Combs
BBN Technologies
10 Moulton Street
Cambridge MA 02138
ncombs@bbn.com

Jeff Vagle
BBN Technologies
10 Moulton Street
Cambridge MA 02138
jvagle@bbn.com

ABSTRACT
In this paper, we identify an agent-based workflow system to mirror critical elements in large systems architectures. We propose the use of this light-weight agent-based service channel to complement existing systems. By pushing the adaptive smarts into critical application connectors, we believe that we can improve the ability of large systems to self-heal and scale.

Categories and Subject Descriptors
D.2.11 [**Software Architectures**]: Languages, Patterns.

General Terms
Algorithms, Performance, Design, Reliability.

Keywords
Agent architectures, Service and Contract workflow, Adaptive Mirroring, Self-healing architectures.

1. INTRODUCTION
A consequence of the difficulty in building reliable software is the increasing use of *system of systems* architectures. One benefit of factoring the business processes into manageable workflow-related chunks is to help moderate the complexity of developing enterprise-scale applications. The promise of loosely-coupled system of systems design is the possibility of building large, sophisticated applications far more quickly (and cheaply) than can be achieved through integrated means.

The Achilles heel of these systems is fixing and evolving them. While commercial solutions are improving the monitoring infrastructure of large systems, they do not solve the more basic problem: current best practices in industry Information Technology (IT) systems extensively rely upon the proverbial "human in the loop" to fix problems.

WOSS '02, Nov 18-19, 2002, Charleston, SC, USA.

Another approach is to automate as much as possible the monitor and repair process. A number of industrial initiatives are based on *self-healing* concepts, notably IBM's "autonomic computing" [1] as well as Sun's N1 initiative [2]. These initiatives are looking to self-configuring, introspective reasoning technologies to manage system complexity. Middleware frameworks can also push adaptive algorithms closer to the components. For example, some middleware approaches (e.g., BBN's *Quality Objects* (QuO) [3]) push QoS decision capabilities to the objects themselves.

One strategy for determining how to repair a system would be to **externally reason** over a complete architectural model of that system. The model could be used to contextualize and filter incoming monitor events; it could identify constraints and properties to validate metrics returned from the target system. Once a problem and a solution have been identified - the architectural model can be used to articulate an effector-based means to implement the repair. For an example of this sort of external approach see [4].

We instead opt for a more reactive approach using localized (imperfect) models of architecture. By **collocating mechanisms** close to the workflow to repair problems as they arise, we believe that we can intervene to solve problems before their effects spread. We use an Adaptive Mirroring technique to insert our agent service within the target system workflow.

We see our approach as complementary to an external approach: a purely reactive approach cannot always work, e.g., addressing certain kinds of deadlock and stability issues may require a more complete picture of the system. A hybrid approach may prove best: use local means first, then punt to an external mechanism for more complete (but arguably slower) analyses and repair.

Our approach, like the external approach, can use *gauges* to push a first round of data consolidation and interpretation closer to their sources in the target system. Gauges are software actors developed by DASADA [5] that can aggregate and interpret probe events.

2. ADAPTIVE MIRRORING
Adaptive Mirroring is based on the use of probes to strategically intercept a target system workflow and circumnavigate data and control through an alternate path. These connectors move data to and from an agent-based *Service Channel* that mirrors the target services. In this parallel *Space,* services are executed and results are returned back into to target system. Because this space has the ability to adaptively shape its internal workflow, we are able to more reliably link data between two or more points within the

target application. This approach is most applicable to distributed collaborative systems (e.g., P2P).

In the DASADA 2002 technical demonstrations [5], we used the GeoWorlds system [6] as a test-bed. By probing the GeoWorlds *Job-Pool*, we were able to intercept the GeoWorlds client software during processing before it reached back to its remote servers – thereby passing data and control through a mirroring system. The GeoWorld's Job-Pool is constructed around a JavaSpace (a JINI [7] service). Our mirroring system essentially mimicked the Job-Pool and linked to the GeoWorlds back-end services using RMI connectors and native components. This mirroring system referenced two core external legacy GeoWorlds services (Keyword Extractor, Noun Phraser).

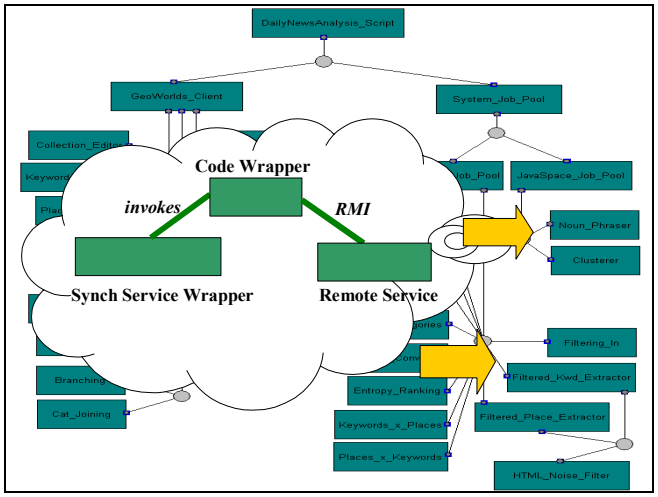

Figure 1. The GeoWorlds client is from the user perspective composed of a number of services related to the Information Analysts job. Two remote service connectors were replaced or "adaptively mirrored" for demonstration purposes Each of these remote connectors in turn are abstractions for a set of lower services.

To demonstrate our ability to work with large sets of interdependent services we created a large pool of variable quality substitutes by wrapping our GeoWorlds components with a *simulation wrapper*. Each substitute component could then be configured to a particular minimum failure rate and invocation latency distribution. In this way a small set of actual components (2) could be "cloned" into a larger number (>50) of components each exhibiting different behaviors.

To date, our experiments have connected to the target system using both RMI and HTTP connectors. More sophisticated connectors based on a QoS JINI variant may be incorporated. In this way, for example, we might be able to roll-up transaction, QoS, and service aggregation (e.g., JINI Service Beans, Operational Strings) semantics from these connector layers into the language expression of the adaptive workflow.

3. DYNAMIC SERVICE SUBSTITUTION

Our self-healing approach is currently limited to the replacement of services and connectors by a mirroring architecture. This assumes that substitutes for failing services (or some dependent) are available.

Our core infrastructure is from an Open Source DARPA-developed agent capability (Cougaar [8]) that has been shown to successfully scale to societies that represented the operations of 300+ military organizations from containing over 1800 domain components (Plugins). Its node-based architecture and component-based design provides scalable flexibility in composing large testable architectures.

Upon this agent infrastructure we have layered a *Service and Contract* (S + C) [9] workflow protocol that can dynamically substitute services in response to runtime performance metrics (see Figure 2). The S + C protocol is a Cougaar styled "language" for wiring up components (*Service Providers*) within a distributed agent-based system. It is based on the publication/subscription of objects on the agent blackboards. These objects represent the semantics of the S + C language: *Requests*, *Acceptances*, *Contracts* etc. As the components are notified and interact via publish/subscribe events – the local assembly of workflow structures is fast, in terms of infrastructure overhead.

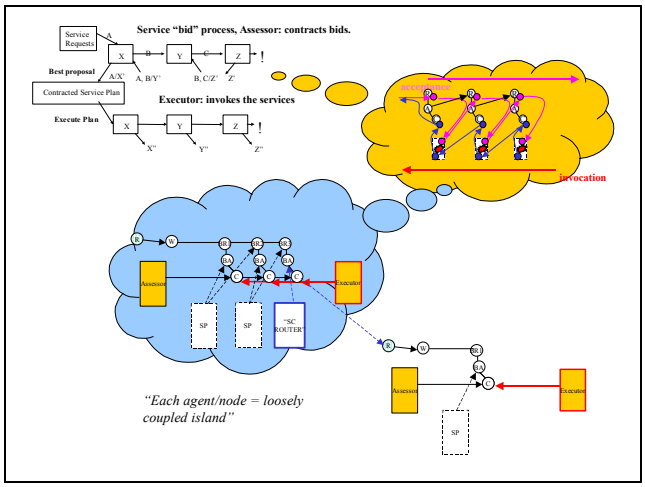

Figure 2 Publish/subscribe events and (distributed) blackboard data model underlies the Service and Contract protocol

Consider an example: an incoming request stimulates a distributed "chain of events" leading to the composition and invocation of a distributed workflow. Services are assembled via an request-accept process: services are **requested**, and Service Providers can agree to **accept**. Acceptance is initially tentative: it isn't until the infrastructure within each involved agent steps in and says, "good, we have a complete set of agreements" that all Service Providers are "Contracted" and invocation commences. The workflow assembly process flows in the forward direction (from the root request outwards). The invocation process flows in the reverse direction (leaves-to-root).

When coordinating component assembly and invocation **across** agents, the S + C design inherits a number of Cougaar computing assumptions, which are perhaps best summarized by: *"agents are widespread and coordination is loose."* Each agent is an island.

Agents complete the piece of the workflow they know about and then solicit for external service providers to fill in for missing services (e.g. service dependencies).

Service Providers (components) are discovered, assembled, and invoked through in reponse to a *Service Request*. *Service Requests* are translated into workflows via the interactions of potentially many services spanning many agents. A workflow represents the service commitments of Service Providers (and all their dependencies) to fulfill Service Requests. Service Requests that are *Accepted* can then be *Contracted* and *Invoked*. Our use of *Contracts* is analogous to other service commitment forms such as *Leases* (JINI) as well as to *Service Level Agreements* (eLiza [10]).

Services are described within the system using an ontology expressed by the DARPA Agent Markup Language [11]. Using DAML services can be hierarchically related. This is useful when matching services at different levels of abstractions. So, for example, a request for a "Search Engine" service might be matched with a "GOOGLE Search Engine" service.

Our approach shares a number of properties with Grid computing architectures [12] as well as to federating JINI-styled architectures. Our use of agent-based learning algorithms is analogous in several respects to the approach used by IBM's ABLE (Agent Building and Learning Environment [13]). Where we diverge from ABLE is in our emphasis on distributed self-organization (the S + C workflow emerges from the distributed and localized interactions of the components).

The S + C's basic adaptive strategy is based on **service substitution**. For example, services that fail may be replaced by the infrastructure if eligible substitutes can be found. In 2001 [5] we demonstrated how Constraints can result in the infrastructure recruiting gauge services on-the-fly to verify some aspect of the operating environment.

Directives are propagated along a self-assembling workflow. Currently, there are two flavors of Directives: Constraints and Hints. One type of Constraint is applied to a service Contract *just before* (PRE) or *just after* (POST) invocation. A side-effect of this may be the recruitment of additional gauge services to satisfy new service requirements [16] (e.g. gauges). Hints, on the other hand, are used to shape the workflow *during* the service assembly process. For example, based on previous experience, Hints may suggest (to the infrastructure) service destinations as well as estimates of a reasonable invocation time associated with a particular service.

We are evaluating use of simple reinforcement learning algorithms to optimize the Directive generation process. For example, Hints may be used to encode error signals based on flowed-back performance metrics. Higher error "temperature" might be interpreted by the infrastructure to encourage more **exploration** (greater tolerance of poorer performing services), and lower temperatures might encourage more **exploitation** (exploit knowledge of previously used "good" services). For more on *exploration-exploitation tradeoff,* see [14].

In the DASADA 2002 technology demonstrations we showed how "information decay" can be used to de-conflict Directives generated at different locations in the distributed workflow. As performance information is "flowed back" it is decayed by the infrastructure as it traverses agent boundaries. Higher "decay" is translated into lower weights for Directives at the point where these Directives are created.

4. ACKNOWLEDGMENTS

This work is sponsored by the Defense Advanced Research Projects Agency (DARPA). In particular we acknowledge the support of Dr. John Salasin and the Dynamic Assembly for System Adaptability, Dependability, And Assurance (DASADA) program. We also thank the Cognitive Agent Architecture Open Source site for providing an excellent infrastructure foundation.

5. REFERENCES

[1] IBM, *An Almaden Institute Symposium: Autonomic Computing*, April 10-12, 2002. http://www.almaden.ibm.com/institute/2002/.

[2] N1: Revolutionary IT architecture for business. http://wwws.sun.com/software/solutions/n1/

[3] Quality Objects. http://quo.bbn.com/

[4] B. Balzer, *Probe Run-Time Infrastructure*, December 2001. http://www.schafercorp-ballston.com/dasada/2001WinterPI/ProbeRun-TimeInfrastructureDesign.ppt.

[5] J. Salasin, *Dynamic Assembly for System Adaptability, Dependability, and Assurance (DASADA)*. http://www.darpa.mil/ipto/research/dasada/.

[6] University of Southern California Information Sciences Institute, *GeoWorlds Project*. http://www.isi.edu/geoworlds/.

[7] The Community Resource for JINI Technology, *Welcome to the New Jini.org*. http://www.jini.org/.

[8] Cougaar Home Page, *Welcome to the Cognitive Agent Architecture (Cougaar) Open Source Website*. http://www.cougaar.org.

[9] N. Combs, Reliable Recruitment and Assembly of Peer-to-peer Services and Distributed Workflow, in *Working Conference on Complex and Dynamic Systems Architecture*, December 2001.

[10] IBM, *eLiza on IBM@server*. http://www-1.ibm.com/servers/eserver/introducing/eliza/.

[11] DARPA Agent Markup Language. http://www.daml.org/

[12] *Global Grid Forum*. http://www.gridforum.org/.

[13] ABLE: A toolkit for building multiagent autonomic systems. IBM Systems Journal, Vol 41, No. 3, 2000

[14] R. Sutton, S. Barto. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, 1998 A Bradford Book