AFRL-IF-RS-TR-2004-225
Final Technical Report
August 2004

ASSURED ASSEMBLY INFRASTRUCTURE (AAI) TOOLKIT

BBNT Solutions LLC

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. K505

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2004-225 has been reviewed and is approved for publication

APPROVED: /s/
JAMES R. MILLIGAN
Project Engineer

FOR THE DIRECTOR: /s/
JAMES A. COLLINS, Acting Chief
Information Technology Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	August 2004	**FINAL** **Jun 00 – Jun 03**

4. TITLE AND SUBTITLE

ASSURED ASSEMBLY INFRASTRUCTURE (AAI) TOOLKIT

5. FUNDING NUMBERS

C - F30602-00-C-0203
PE - 62301E
PR - DASA
TA - 00
WU - 04

6. AUTHOR(S)

Nathan Combs

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

BBNT Solutions LLC
10 Moulton Street
Cambridge MA 02238

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington VA 22203-1714

AFRL/IFTB
525 Brooks Road
Rome NY 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

AFRLIF-RS-TR-2004-225

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: James R. Milligan/IFTB/(315) 330-1491 James.Milligan@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***

This technical report describes work performed on a project sponsored by DARPA/IPTO's Dynamic Assembly for Systems Adaptability, Dependability, and Assurance (DASADA) Program. Specifically, this project developed the Assured Assembly Infrastructure Toolkit (AAIT) which provides a collection of software and documented techniques to test a uniform assembly model for integrating heterogeneous system components. The AAI Toolkit also explicitly models Gauges that measure and drive the dynamic assembly and reconfiguration of the software architecture. Through "service contracts" and real-time feedback, the AAI dynamically adapts system architectures to optimize system performance with respect to performance metrics.

14. SUBJECT TERMS

DASADA, software composition, probes, gauges, service contracts, architecture

15. NUMBER OF PAGES

72

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

TABLE OF C O N T E N T S

List of Figures

1. Introduction

1.1 Project Overview

The Dynamic Assembly for Systems Adaptability, Dependability, and Assurance project, (DASADA) was sponsored by Defense Advanced Research Projects Agency (DARPA), with the Air Force Research Laboratory (AFRL) serving as Lead Technical Agent. As part of the DASADA project, the team of BBN Technologies and JXML Inc., has developed an Assured Assembly Infrastructure (AAI) Toolkit that realizes software workflow architectures that can dynamically adapt based on specified performance objectives.

The AAI Toolkit provides a collection of software and documented techniques to test a uniform assembly model for integrating heterogeneous system components. The AAI Toolkit also explicitly models Gauges that measure and drive the dynamic assembly and reconfiguration of the software architecture. Through "service contracts" and real-time feedback, the AAI dynamically adapts system architectures to optimize system performance with respect to performance metrics.

The AAI Toolkit uses a dynamic assembly mechanism for constructing software architectures of software components and Gauges (DASADA [4]). It uses an adaptive workflow to reconfigure its architecture. It uses XML to bridge multiple-levels of description (metadata, architecture, and software).

The AAI Toolkit consists of these elements and characteristics:

- *Components*: Can easily add new plugins to extend the AAI Toolkit infrastructure or applications built using it.

- *Services and Contracts*: Services and Contracts identities a workflow protocol that defines the interaction of components. It consists of an event-based language specification and infrastructure assumptions about how system requirements and component dependencies are negotiated.

- *Assessors and Routers*: Assessors and Routers comprise an infrastructure that interacts with Services and Contracts to perform requirements tradeoffs and drive creation of assemblies of components and Gauges.

- *Architecture Model*: The Architecture Model is an external representation of the assembly of components and Gauges within the system. It serves as a representation that specifies the target system behavior and as a model of the actual form of a system (time varying).

- *Executors*: Executors invoke the assemblies of components and Gauges to realize the specified software behavior.

- *Gauges*: Gauges are a DASADA sensor type that provides constant feedback to the infrastructure so that it can composes/reconfigure itself to better match the requirements.

- *Dynamic Adaptation of Services and Contracts.* This feature allows Components and Gauges to be moved into and out of execution assemblies based on performance and changing application requirements.

Over the course of this project we demonstrated the AAI Toolkit with two distributed web-service applications. We showed how this technology can reconfigure itself from Gauge feedback.

1.2 Final Research Products

This project delivers a means for flexibly testing of distributed adaptive architectures. This AAI Toolkit consists of these research products:

- A protocol for building and modifying assemblies of distributed services (workflow).
- An agent-based infrastructure for instantiating the workflow over a real implementation.
- Two demonstration applications illustrating core features of this research.
- An *EventAtlas* for converting events from the distributed infrastructure into Architecture Description Language (ADL) form.
- Procedures and algorithms for adapting systems compatible with the described protocol and the agent-based infrastructure.

2. Approach

2.1 Motivation

Building large, reliable software systems is difficult and expensive. One practice for managing such an effort is to use a *system-of-systems* (SOS) architecture. An SOS architecture makes the development of enterprise-scale applications easier, primarily because loosely-coupled systems are easier to engineer and maintain than more traditional, integrated designs.

However, the Achilles heel of SOS systems involves the difficulty of repairing them in operational settings. Often SOS systems rely upon a variety of middleware and monitoring capabilities that may transcend multiple administrative and work groups. Currently available approaches tend to rely extensively upon the proverbial "human in the loop" to fix problems as they arise during operation.

The objective of our research was to identify supporting capabilities that can coordinate service discovery, connector management, architecture monitoring, as well as provide access to mechanisms for remedial actions – e.g., tune Quality of Service (QoS). We would contrast this to an SOS solution that either relies upon extensive monitoring or one that is a "crazy quilt" of individual solutions stitched together.

Our objectives stand in contrast to a "monitor-everything" and "fix it by operator" approach. Why? First, the human in the loop is slow and expensive. Second, setting up monitor probes by hand is hard to do correctly and is unlikely to be done correctly by many different developers. Furthermore, the sources of many problems will involve elaborate traces through many levels. Building monitoring conduits through large SOS implementations that link large numbers of users, processes, components, and domain models is difficult to do correctly. This leads to the third problem: it is hard to balance the *quality* vs. the *quantity* of information in large systems (too much or too little of either can be an impediment to making timely decisions).

We contrast our approach also with typical Department of Defense and industry practices for managing system reliability. These practices are mainly focused upon developing and standardizing middleware component frameworks, or upon constructing operator-intensive processes for monitoring systems. In this research we sought a capability for managing and evaluating QoS concerns across a range of application and component granularities. Our approach is consistent with middleware solutions. Our service-based approach is agnostic to particular middleware approaches - this is particularly useful if one believes that no particular middleware solution can scale to all SOS systems (notably legacy systems). Our focus was upon the organizing principles for diverse community components to interact within an architecture (vs. individual interoperability, for e.g.). This provides a good foundation to think about collective behaviors and metrics within an application.

Furthermore, our service-based approach can flexibly integrate feedback from external sensor sources in a number of ways. In this regard, a particular style of use developed by DASADA was demonstrated. Gauges and Probes were individually developed by the DASADA community, and their use was illustrated in two annual demonstrations. The DASADA community also specified a Gauge-and-Probe infrastructure (Common DASADA Infrastructure: CDI [3]) with which the AAI Toolkit was compatible. Integrating external sensor grids such as CDI with the

AAI Toolkit can offer other synergies. For example, bridging the services oriented architecture view of the AAI Toolkit with an object-level adaptive QoS middleware that can span a variety of platform and communication protocol spaces (e.g., [2]) could be lucrative.

2.2 Architecture

Our approach started with a number of research building blocks. Our goal was to look at this problem synergistically (whole greater than the sum of the parts) starting from these building blocks:

- A service-oriented software model.
- A reactive architecture model that distinguishes between low-level "autonomic" responses and high-level execution planning.
- An adaptive model where DASADA Gauges can control how connectors and services are configured and used.

Knitting these building blocks together were software agents. The collaborative exchanges of these agents and these building blocks instantiates a dynamic service architecture.

We based our implementation on an Open Source DARPA agent infrastructure called Cougaar (www.cougaar.org). Our use of Cougaar served two purposes. First, it identified a specific implementation that is sufficiently flexible for building and testing ideas. Second, it provided a mature platform upon which to demonstrate our results.

2.3 Scope of the Experiments

In 2001 we demonstrated the core Service and Contract (SC) ideas using a webservices application (DASADA Technical Demonstration, Baltimore, [5]). We prototyped an *Abstract Query Engine* application. The Abstract Query Engine performed text-search, web-scraping, and database/query services for an Information Analyst tool.

In 2002 we demonstrated a *SmartChannels* application of the AAI Toolkit (DASADA Technical Demonstration, Baltimore, [5]). The SmartChannels demonstration provided a "fail-safe" capability to an Information Analyst tool by monitoring critical connectors to remote services and intervening as needed. The SmartChannels system would route data to and from an alternate set of substitute or back-up services in lieu of the failed services.

Through these experiments we were able to demonstrate:

- *A system of software agents that can replace services (substitution) and connectors (alternative pathways).*
- *Software agents that can adaptively enhance their performance and the reliability of the workflow they instantiate.*

2.4 A Local Strategy of Reactive Repair

One approach for detecting errors and repairing systems would be to collect events then derive measurements of the running system and compare them against an external model of that system. In this *external* approach, solutions could be injected into the system via effectors [3,4]. The key here is to build a global picture of the system at a particular moment in time, and then to use it to decide a course of action.

We pursued a contrasting approach based on accepting less-than-perfect information and reacting to events and metrics close to their source. By forfeiting access to the larger picture of the architecture and the rest of the system, the research objective was to trade-off speed with understanding. Thus, the techniques described in this report emphasize *collocating mechanisms close to the components and connectors and repairing problems as they surface*. The payoff is that our approach can intervene to solve problems before their effects spread.

We see our approach as complementary to the external approach. In fact, a purely reactive approach such as ours cannot always work, e.g., addressing certain kinds of deadlock and stability issues can require a more complete understanding of the system. Thus, a hybrid approach may prove best in the long run; that is, when possible, fix problems quickly using local information and mechanisms, otherwise use an external reasoning system that can handle more complete (but arguably slower) analyses and repair.

2.5 Other DARPA Research (Leveraged)

Our objective was to test against complete prototype systems. We were able to do this by leveraging existing research and work. For example, we leveraged extensively from the Open Source community. We also incorporated other DARPA research into our technology mix, namely the Cougaar Open Source agent framework and the DARPA Agent Markup Language (DAML [16]).

3. Technical Discussion

3.1 Agents

AAI Toolkit agents are lightweight software entities that can encapsulate (or can control) component services. Agents coordinate amongst themselves to execute their constituent services. Agents monitor their constituent services and receive metrics and other event inputs from other agents or sensors (e.g., DASADA Gauges). And finally, agents also monitor their progress and learn to improve their own future performance.

Cougaar [8] Open-Source software was our underlying agent framework. With Cougaar we were able to leverage a mature and well-maintained software codebase. Additionally, Cougaar's proven scalability meant that we were able to deploy and exercise sizeable testbeds (> 18 agents, > 100 components in 2002). Larger system testing gave us greater confidence in our ability to generalize our research results. For example, it wasn't until we started working with larger numbers of agents in 2001 that we realized the need to flesh out the concept of a service dependency neighborhood that surrounds each component. This neighborhood varies by availability of information (to the agent owning the component) as well as by the importance of the task. More important tasks can demand larger effort by the system – e.g., casting a wider net to insure that all dependent components are found. Without a sense of neighborhood and a set of policies for guiding how to work within and without that neighborhood, agents can saturate the communication bandwidth of the system by excessively messaging each other (acting greedily). This understanding, especially in the context of the SC protocol and infrastructure, was easy to overlook unless working with a larger testbed.

Another important aspect of our use of Cougaar agents was that we wanted to be able to factor the design and ideas of the SC protocol from the implementation. Beyond its small footprint, Cougaar makes few demands upon the design pattern beyond a Plugin component model, use of publish/subscribe events, and the use of object replication rules for messaging between agent Blackboards. The simplicity of these design elements, as well as their usefulness in building more complex and sophisticated protocols, allowed us to develop a design that is portable to other frameworks. Other agent-based paradigms are compatible with our approach. For example, we also examined the use of an OMAR agent framework[1] instead of Cougaar.

By using Cougaar we were also able to contrast our research with other planning work conducted within other Cougaar communities; e.g., from the logistics planning domain (Ultra*Log [6.]). In this way we were able to leverage previous experience. For example, some of the ideas surrounding our design of "Executor Plugins" and "Assessor Plugins" (detailed later in this report), were inspired by prior Cougaar work.

3.2 Event-based Service Collaboration Language

AAI Toolkit agents collaborate amongst themselves as well as negotiate internally with their components and infrastructure using an event-based "language". This language is centered on publish/subscribe events. Events are contextualized by data objects that constitute a Logical Data Model (LDM). For example, publishing a Request data object on the local agent Blackboard indicates that a service is sought. Responding to this event, a Service Provider may then issue an Acceptance event (publication of an Acceptance object onto the Blackboard). Thus, from a

single event a cascade of other events (other services being sought, Assessors and Executors coordinating actions, etc.) can flow.

Language elements may be replicated across agent boundaries according to well established rules; and in so doing, stimulate responses and activities throughout the system. The combined interactions of components, infrastructure, and agents are defined by a protocol – the Service and Contract protocol (described in detail later). So, for example, a Request object that fails to solicit a service response within a local agent (e.g., a service provider not responding to the request), may be replicated on other agents where responders may be found - subject to the rules of the protocol. In Figure 1 we see how a Request issued from a User Interface (UI) to an SC agent can lead to interactions among two agents. In this case only two of the required three services reside in the first agent. Through the interactions of the components and the infrastructure via the SC protocol, events are integrated into a unifying workflow distributed over two agents.

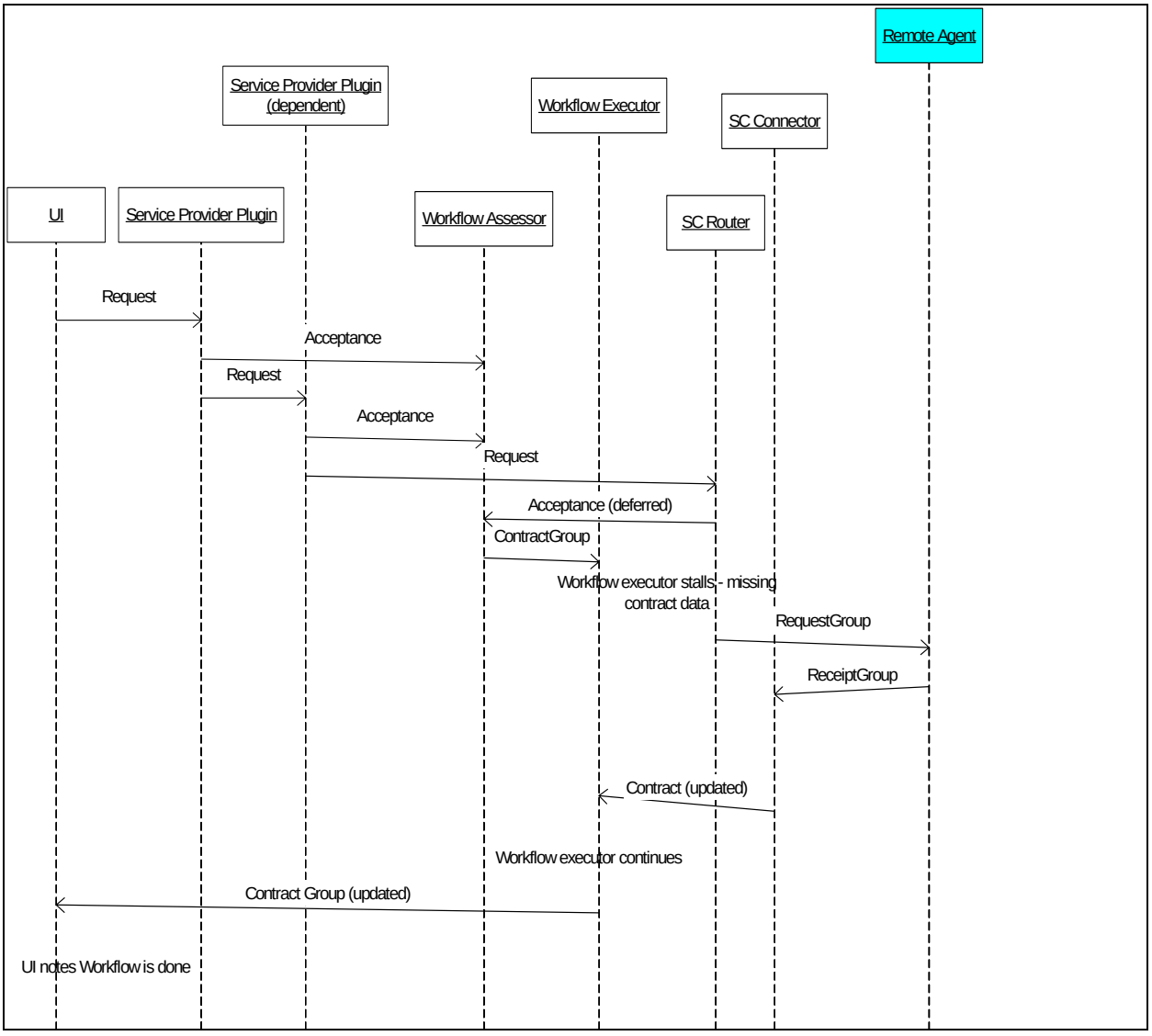

Figure 1: Example of the component interactions within a single agent based on the Service and Contract publish/subscribe "language".

3.3 Component-based AAI Toolkit Infrastructure

Composition of SC systems using the AAI Toolkit is based on the Cougaar Plugin model. Plugins in the AAI Toolkit come in two flavors. Some Plugins are infrastructure components that are used by the agents to encapsulate services used to help them organize planning and execution of services. These are called *Infrastructure Plugins*. Another flavor of Plugin is *Domain Plugin*s. Domain Plugins integrate application services (e.g., application code, gateways to web services, etc.). Domain Plugins do the work that constitutes the application.

The AAI Toolkit infrastructure as implemented, based on Cougaar, only interacts with Java™ Plugins directly. However Plugins can interoperate with external processes via Java Native Interface or over socket. Using the AAI Toolkit plugin model we can easily integrate other DASADA-developed components such as Gauge and monitoring services. For example, in the 2003 DASADA Technical Exposition, we demonstrated how a *DASADA Gauge message bus* could be integrated into an AAI system simply by swapping in Plugins that are able to communicate with the Gauge message bus using Java Remote Method Invocation (RMI) [27]. The DASADA Gauge message bus was a service that could potentially handle messages from a grid of Gauges or other sensor sources.

For a more detailed examination of the compositional flexibility afforded by Plugins, see the *Supporting Investigation* sections of this document. Our inheritance of the Cougaar styled text-configuration files to populate agents with Plugins enabled us to easily swap components for testing and experimentation.

```
plugin =
com.bbn.openzone.base.plugins.DAMLConceptManager(daml=TEST.app.daml,implies_ou
t=false)

plugin =
com.bbn.openzone.base.plugins.SCRouter(COMMS=USE_YP,MAX_SENDREQUESTS=1,CONCEPT
NAME=ROUTER_SERVICE)

plugin = com.bbn.openzone.base.plugins.SCConnector

plugin = com.bbn.openzone.base.plugins.WorkflowRegulator

plugin = com.bbn.openzone.base.plugins.WorkflowAssessor

plugin = com.bbn.openzone.base.plugins.WorkflowExecutor
```

The core AAI Toolkit infrastructure Plugins are given below – they are used by agents to implement the Service and Contract workflow protocol. These plugins are described in greater detail in a later section.

- Executor Plugin
- Assessor Plugin
- SCRouter Plugin
- SCConnector Plugin

Other types of AAI Toolkit infrastructure Plugins include the WorkflowRegulator and the OntologyManager (or DAMLConceptManager as actually used in the software). These are not core to the implementation of the SC protocol but are necessary to making the infrastructure work within the developed AAI Toolkit.

Domain Plugins can participate in an SC system so long as they provide the following interfaces:

1. The Cougaar Plugin interface (See Cougaar Developer's Guide [8.])

2. The AAI Toolkit Service Provider interface (see the SC Developer's Guide [9])

All Infrastructure Plugins are Cougaar Plugins. Infrastructure Plugins may implement the AAI Service Provider interface, depending upon the Plugin's role. For example, because the SCRouter serves as a broker for remote services within an agent and is able to Accept Requests on behalf of a remote service, an SCRouter appears to the infrastructure as a ServiceProvider. Thus, as a ServiceProvider, the SCRouter can Accept Requests using the normal mechanisms. In contrast, the Executor and the Assessor are not Service Providers… because in their roles they have no need to Accept Requests.

To simplify the implementation of Service Provider components within the AAI Toolkit system, a base ServiceProviderPlugin class is provided. Any Plugins extending this base class will be known as Service Providers within an SC Cougaar system.

```
package com.bbn.openzone.core.plugins;

/**

 * Base class from which Service Provider PlugIns (domain) can

 * use (extend) for basic behaviors.

 */

public class ServiceProviderPlugIn extends OpzSimplePlugIn implements
ServiceProvider
```

The ServiceProvider interface requires that the Plugin be able to declare its service type. Service types are defined using a DAML-based ontology. Each Agent (and all the services owned by it) are described by an ontology. Agents may share an ontology – but they are not required to do so. To be useful, a ServiceProvider should be able to subscribe to Requests on the Blackboard that are relevant to it. Presumably, the ServiceProvider would then be able to examine these Requests and Accept some of them based on some internal evaluation. Implicitly, the ServiceProvider would upon invocation provide some service relevant to the Request.

An example of a simple ServiceProviderPlugin that performs rudimentary Request/Acceptance bookkeeping is provided in Figure 2.

```
/**
 * A MOST BASIC SP PATTERN PLUGIN
 * Which launches Dependency REQUESTS when notes a Request which matches
 * its request.
 */
public class TestSPPlugIn extends ServiceProviderPlugIn
{
      private List myDependencies = new ArrayList();

      public void setupSubscriptions() {
          // super.setupSubscriptions() must be called
          super.setupSubscriptions();

          myDependencies = getAllStringParameters(getParameters(), "DEPENDENCY=", "");

          // setup subscription for Requests, noArg = use default SP predicate
          setRequestsSubscription();
      }
      public void execute() {
           // super.execute() must be called
          super.execute();

           List myNewRequests = getOutstandingNewRequests();
           Iterator it = myNewRequests.iterator();
           while( it.hasNext() ) {
              Request req = (Request)it.next();
              //System.out.println("[SP, ID=" + this.getPlugInID() + "> received request: " + req + "]");
              BlackboardService bb = this.getSubscriber();

              //
              // accept() implicitly publishes Acceptance and any dependent Requests
              // if no dependencies, myDependencies is empty list...
              Acceptance ac = accept(req, myDependencies, Relationship.AND, bb, req.getData() );
           }
      }
      public void invoke(List importsBindings,  Acceptance accept, DataConnector exportDC) throws NonInvocableContractException
      {
          System.out.println("-----------------------------------------------------");
          System.out.println("[TestSPPlugIn] accept.getParent().getData().toString()=" + accept.getParent().getData() );
          System.out.println("-----------------------------------------------------");
      }
}
```

Figure 2. A minimal example of what it takes to "code" a Service Provider Plugin – it is a Cougaar styled Plugin that accepts matching Requests and issues a dependency Request (from Plugin parameters). Its invoke() method is stubbed - an actual domain Plugin would provide implementation.

ServiceProvider Plugins in the 2001 and 2002 SC systems were demonstrated as short-lived services. A short-lived service is a service whose invocation is characterized as being of a short duration at whose completion a result is returned (including NULL result). Note that from the perspective of the infrastructure, a short-lived service that is serviced by a long-running process in the background is indistinguishable from a call to a Plugin in the same process. (A short-lived service is in contrast to a long-lived service, whose results might be streamed over a long period of time.) We conducted preliminary design work (in anticipation of a DASADA Phase-2 effort) into extending the SC design (in the same extensible manner as the Reliable Multicast Framework experiments described in later section) to support streaming connectors and workflow. This would have enabled handling of long-lived services within an SC system.

3.4 Service and Contract Protocol

Our core infrastructure comes from an Open Source DARPA-developed agent capability (Cougaar [8]) that has been shown to successfully scale to societies that represent the operations of 300+ military organizations and contain over 1800 domain components (Plugins). Cougaar's

node-based architecture and component-based design provides scalable flexibility for the composition of large, testable architectures.

The SC workflow protocol manages the dynamic response of a workflow of services to runtime performance metrics. The SC protocol is constructed from a Cougaar-styled “language” for wiring up components (Service Providers) within a distributed agent-based system. The vocabulary of the language consists of the events related to the publication and subscription (publish/subscribe) of objects onto the local agent Blackboards. Figure 3 illustrates the Service and Contract Logical Data Model (LDM).

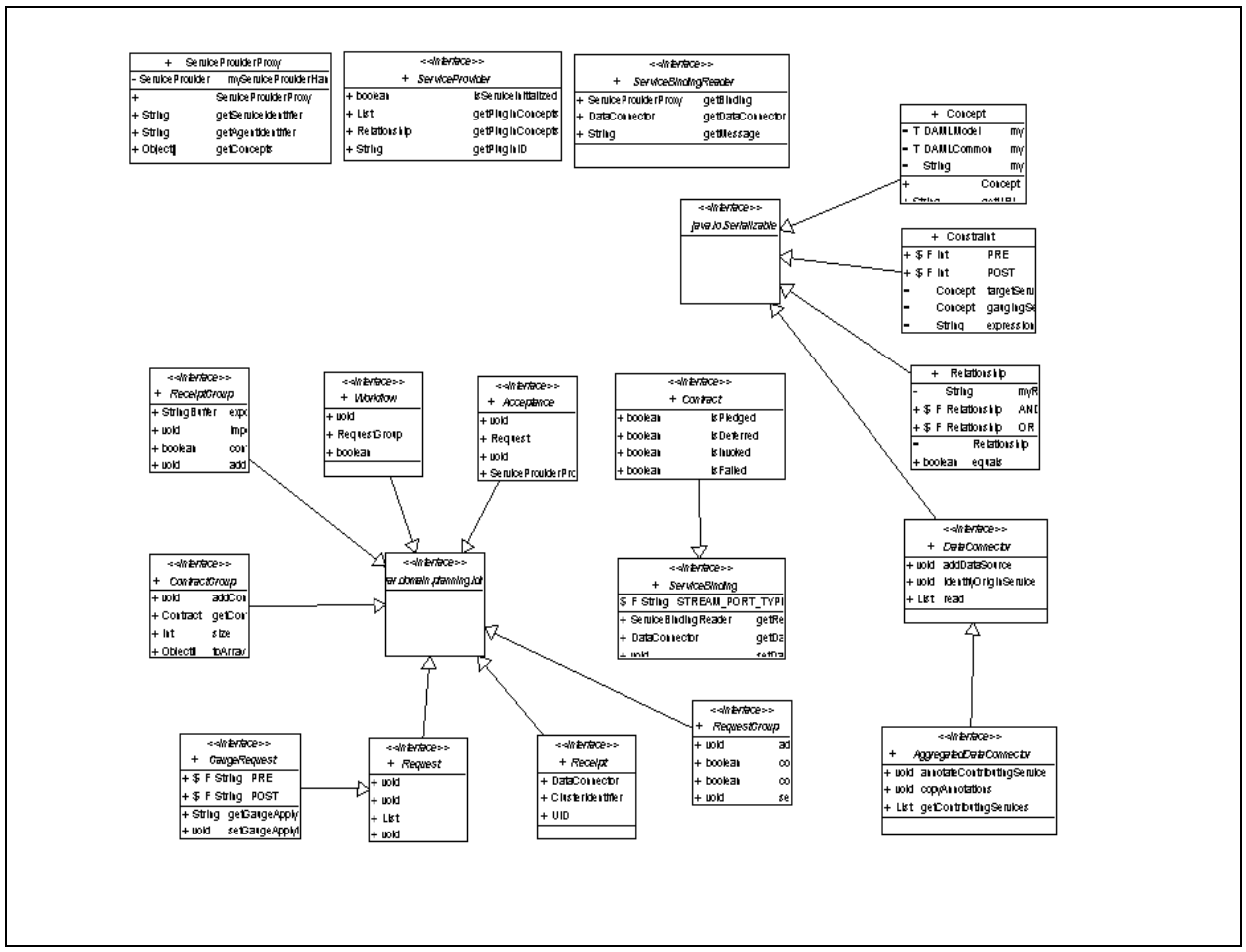

Figure 3. Core Service and Contract Interfaces (Logical Data Model: LDM).

As illustrated, the following classes are key Service and Contract language elements: *Request*, *Acceptance*, and *Contract*. Instances of these classes (as well as other LDM elements) are used to contextualize a component pattern of publish/subscribe (events). The combination of the objects (language) and the pattern of events define the Service and Contract protocol.

Because the components and infrastructure are reactive (communicate via publish/subscribe events), and because the SC protocol is largely parallel, the actual assembly and invocation of workflow structures from an infrastructure perspective is fast.

Consider that an incoming Request stimulates a distributed "chain of events" that leads to the composition and invocation of a distributed workflow. Services are assembled via a *request-accept process*: services are requested, and Service Providers can agree to accept. Acceptance is initially tentative; after the infrastructure within each involved agent acknowledges the existence of a complete set of agreements, then all Service Providers are *contracted*, and invocation commences. The workflow assembly process flows from the root request outwards ("forward"). The invocation process flows in the reverse direction (leaves-to-root).

When coordinating component assembly and invocation *across* agents, the SC design inherits a number of Cougaar computing assumptions, which are perhaps best summarized as follows: *"Agents are widespread and coordination is loose."* To understand this concept, think of each agent as an island. An agent partially completes a workflow and then solicits for an external service provider (another agent) to fill in for missing services (e.g., service dependencies).

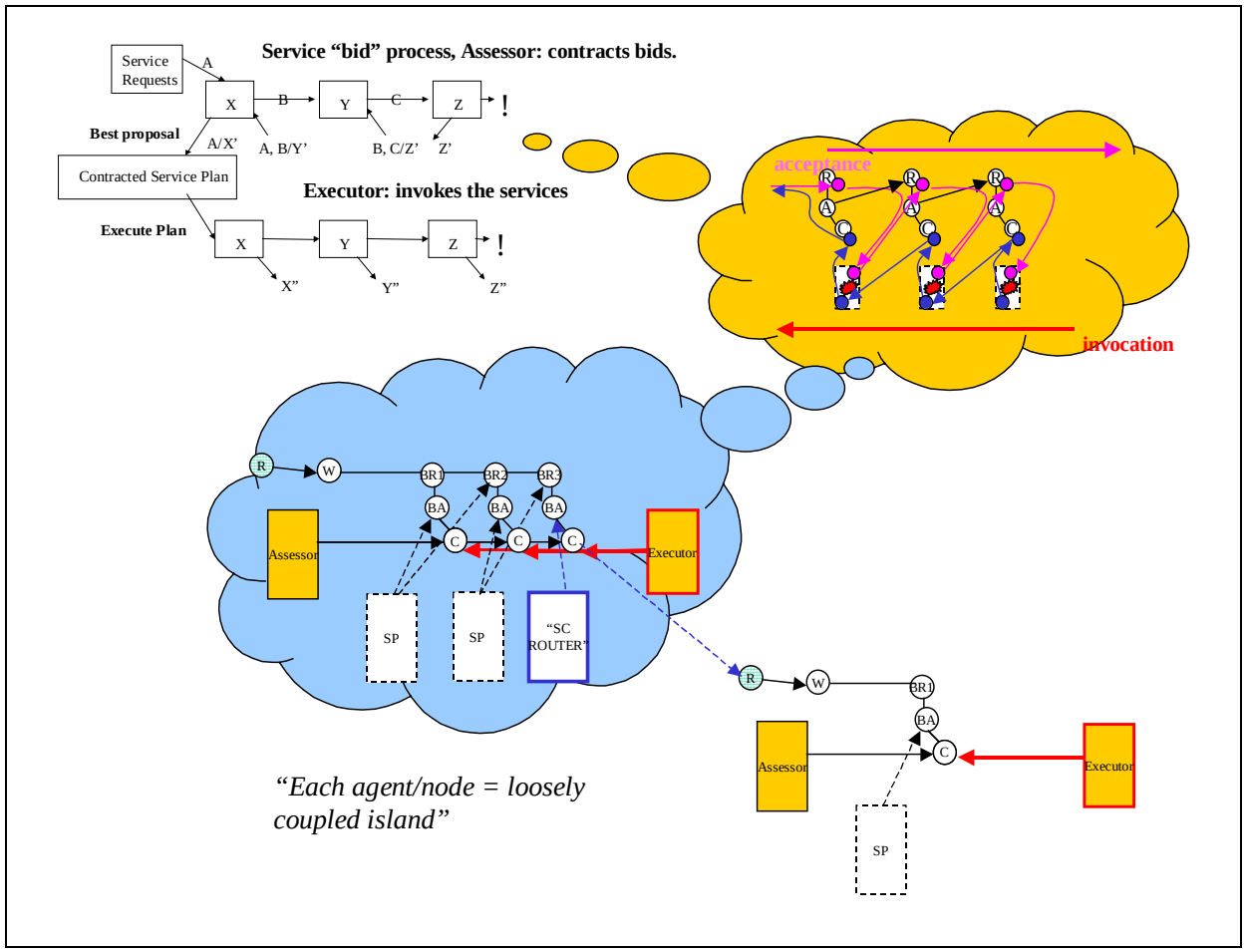

Figure 4 Publish/subscribe events and (distributed) Blackboard data model underlies the Service and Contract protocol.

Service Providers are discovered, assembled, and invoked by a *Service Request*. A Service Request is converted into a distributed workflow via the interactions of at least two, but potentially many more services spanning many agents. Services are pledged by *Service Providers* (components). A Service Provider can be a proxy for an external process, a service, or

an entire system. Services are then *contracted* and *invoked*. The use of *Contracts* within the workflow is analogous to other service commitment forms such as *Leases* (in JINI) as well as to *Service Level Agreements* (in eLiza [33]).

The SC language differs from other Cougaar languages (e.g., the military-logistics version [6]) in part because of its reliance on specialized infrastructure services (implemented as Cougaar Plugins) that enforce a more structured workflow protocol (e.g., explicit assessment, contracting, and invocation stages). Packaging of infrastructure services into Plugins makes it easy to bootstrap sophisticated behaviors from a basic set of building blocks.

Services are described by a Resource Description Format [15] language: the DARPA Agent Markup Language [16]. One benefit of DAML that we exploit is that services can be hierarchically related in the service ontology. 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.

In the next section we’ll discuss more fully the form of the SC protocol.

3.5 Service Hypothesis (Plan) vs. Service Execution.

The SC protocol enables instantiation of SC workflows whose effect is to coordinate services first by *Plan* and then by actual *Execution*. To instantiate an SC workflow within a single agent minimally requires a pair of infrastructure Plugins: the *Assessor* and the *Executor*. The Assessor is the capstone in the process by which the invocation of Service Providers is planned [(A.) + (B.) in Figure 5]; the Executor dominates the process by which once Service Providers are Contracted, they are actually executed [(C.) + (D.) in Figure 5].

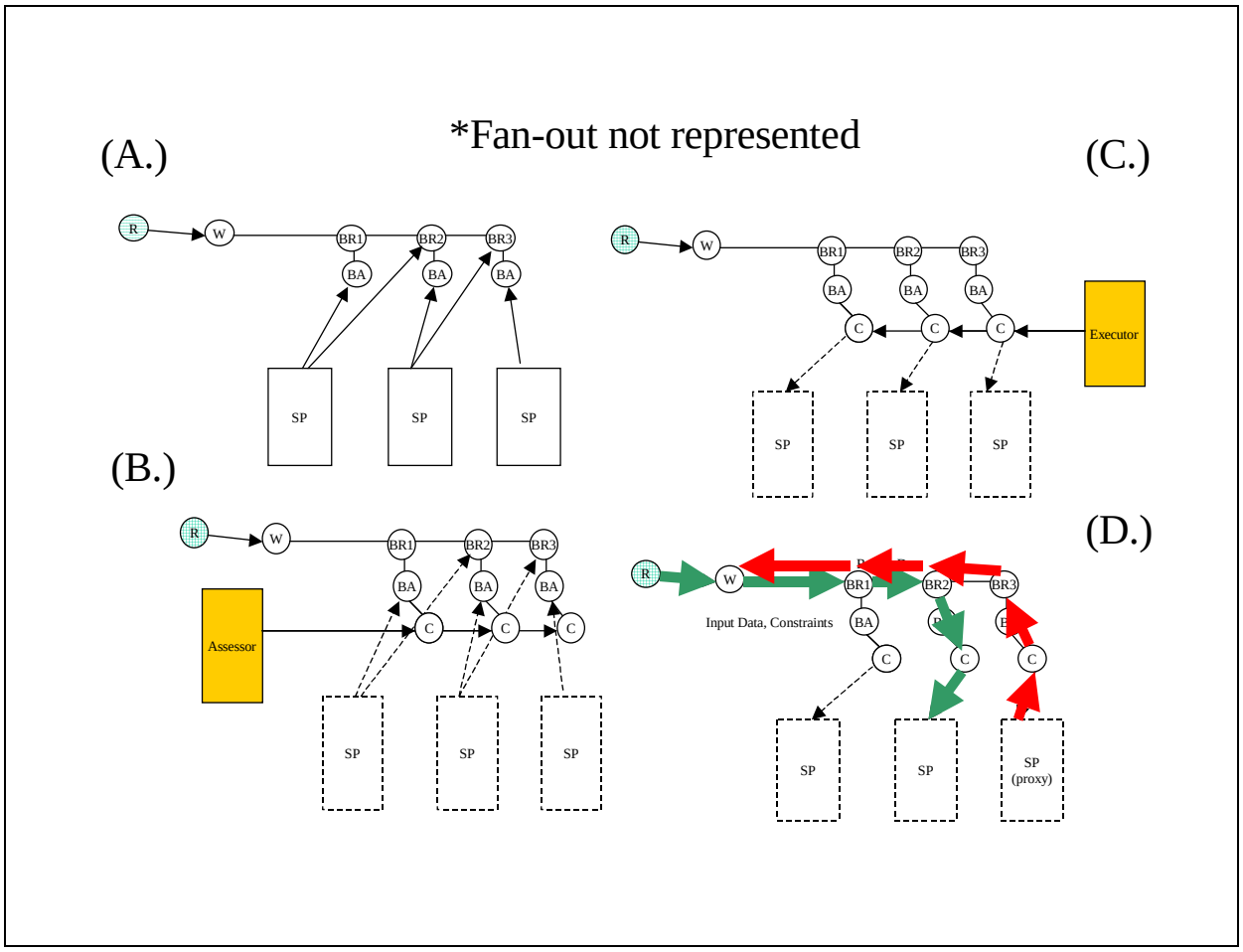

Figure 5 The Basic Service and Contract workflow

As we see in Figure 5:

- (A.) Service Providers PlugIns (labeled “SP”) tentatively accept requests. A “service chain” is constructed as Service Providers request additional dependency services, etc.
- (B.) At each node, an Assessor infrastructure PlugIn monitors progress. When the Assessor is convinced the Service Chain is complete, the Assessor steps in and “Contracts” service chains.
- (C.) The Executor notes when a Service Chain has been Contracted. The Executor then “invokes” services in reverse order.
- (D.) A Service and Contract workflow serves as an “information network”. In the forward direction, data and constraints are propagated. In the reverse direction, results (service invocation) are propagated. Note that Service Providers “in the middle” of a Service Chain can monitor (and potentially change) data and results as they flow “to and fro”.

The tool of the Assessor is the *Contract*. An Assessor encapsulates the rules and capabilities to evaluate the Acceptances issued by ServiceProviders and to select the ones it wishes to commit to action. For example, multiple ServiceProviders may offer their services (issue an Acceptance) in response to an incoming service Request. The Assessor chooses one and awards a Contract.

In the 2001 and 2002 demonstrations, the Assessor used simple rules to evaluate Requests. It essentially looked for the first *Well-formed Request* and issued a Contract. A Well-formed Request was one that had all its dependencies Contracted and that had no outstanding and unaddressed Constraints.

Once the Assessor has Contracted a complete service branch (within the local agent), the Executor then invokes those Contracts. The design objective is to defer invocation of a service branch until all services (dependencies and constraints) have been Planned. Thus during the assembly of a service, workflow conflicts and constraints can be resolved before services have been actually invoked.

A number of interesting questions were examined – for example, whether an Assessor explicitly can hedge its bet and award multiple Contracts for competing sibling services. It turns out that it can, but it must explicitly choose to do so. It can do so, for example, as an insurance against a single service not working out. But it must weigh the implication of the extra work incurred by the system (multiple competing threads of services).

Within an agent, the planning and execution steps are handled differently. Because different branches of the workflow may be at different levels of maturity and because agents can only loosely coordinate, the planning step may occur while execution is occurring elsewhere. Service Providers interact with the workflow (Accept) asynchronously and at their own pace (how busy are they?)– meaning that rates of development of the workflow may vary within the system. In contrast, the execution step is serialized. Sequential execution occurs once Contracts have been issued (by the Assessor) – there is a single Executor that walks through the outstanding Contracts within any given agent.

A design evaluation was undertaken on extending the SC protocol to permit multiple Executor Plugins operating in parallel within a single agent. The conclusion was that it is possible with some adjustment to the SC protocol so that Executor Plugins can communicate amongst themselves within a single agent - to coordinate actions. An alternative approach is the one adopted in the 2001 and 2002 demonstrations. In these demonstrations, parallel execution of services was managed by partitioning services into multiple agents (vs. a single agent with multiple services). As each agent operates independently, where branches split across agent boundaries, parallelism occurs. This suggests an interesting research question: what is the proper granularity of *agent vs. service*, and how can we *quantify* this relationship? Should agents encapsulate many services or should there be many agents? Ultimately we feel the answer depends upon the application and the granularity of the service/components.

When assembling services over distributed nodes, an additional infrastructure Plugin is required (SCRouter). The SCRouter loosely coordinates the workflow between nodes – occupying an interesting research niche. Over the life of the BBN project, the SCRouter has evolved into a

futures broker for services. Specifically, it has become an SC infrastructure actor that places intelligent *bets* about the availability of remote services.

Nominally, the SCRouter is an infrastructure component that watches the local Blackboard for unsatisfied Service Requests and decides what to do. In other words, if a service is not found locally, it ends up on the Blackboard as a *Dangling Request* (a Request no one has Accepted). At this point the SCRouter may choose to send the Dangling Request afield (typically) to other agents who might have services available that can satisfy the Request. How does it guess which remote agents might have available relevant services? When the SCRouter sends a Dangling Request to a remote agent, the SCRouter is essentially performing a bet on behalf of the infrastructure at the local agent. In order for the local agent infrastructure to stabilize around a workflow hypothesis/plan, the "unfulfilled" Request must be Accepted (and subsequently Contracted) by someone. In the case of a Dangling Request this is performed by an SCRouter who is acting on a sort of "bet" that it can find a remote provider: it Accepts the Request and then sends a copy on.

In 2002 the implementations of the SCRouter used a three-tier algorithm when deciding where to send Remote Requests. First, the SCRouter looked at historical performance data (past workflow metrics), then it looks at a Yellow Pages service (if it exists), and then failing the above it broadcasts to the local neighborhood.

The Service and Contract protocol is designed for a large distribution of agents where global synchronization of the activities of agents cannot be practically enforced, because to do so would either require insertion of a new infrastructure piece that can act as a central coordinator, or it would require a more elaborate plan negotiation phase. The latter option has been considered and preliminary design work has been completed.

Every agent owns its own copy of the Service and Contract infrastructure components (e.g., Assessor, Executor, SCRouter,…). In other words, there is no global Assessor, Executor, etc. One consequence of this is that each infrastructure Plugin has visibility into only a piece of the workflow; i.e., the piece of the workflow that resides on the local Blackboard. Visibility into the activities of other agents is provided only to the extent they propagate SC LDM objects (Requests and Receipts). Replicated objects become "cues" that are translated into the local vernacular: local LDM objects and publish/subscribe events.

The SC patterns presented here are exactly descriptive of behaviors within a single agent. In the case of where services are distributed among multiple agents, additional qualification is needed. The current SC design inherits a number of Cougaar computing assumptions, which are perhaps best summarized by the following rules:

Agents are widespread and coordination is loose.

While over time we are likely to modify some of these assumptions (optionally) to more exactly enforce the SC patterns in a distributed environment, it is worth exploring the current impact.

Assessment and invocation are locally controlled. There is no such thing as a global assessment or invocation step.

Think of each agent as an island. An agent partially completes a workflow and then solicits an external service provider (another agent) to fill in for a missing service (e.g., a dependency).

When Requests are sent to external agents there is no direct coordination between Assessors and Executors across agents. It is up to every agent to evaluate and invoke services according to its own rules and policies. While this may lead to local choices that are in conflict with unstated global preferences, it is best left to the owners of services to judge and manage application of their services.

As we described earlier, constraints are propagated with the workflow and may be used to direct Assessors and Executors. An important distinction, however, is that without global control there is no mechanism of enforcement. It's up to the local parties to "do the right thing."

There is no infrastructure-level synchronization of assessment or invocation of services.

The SC infrastructure borrows from the Cougaar design philosophy that a large-scale synchronization of workflow elements is not scalable across large and widely dispersed systems. In the current infrastructure, this point is related to the following:

There is no guarantee that another agent will notify you of what it did.

This loose-coordination assumption is explicitly enforced in the SC protocol via these aspects of the design of the system:

1.) When a Service Provider accepts a Request – the Acceptance is a tentative commitment. It isn't until the Assessor (infrastructure) Contracts this Acceptance that this commitment is considered binding and recognized by the Executor (infrastructure). Once an Acceptance is Contracted, the Executor can invoke the underlying service.

2.) Only after a Request has been Accepted and Contracted to an SCRouter can it be sent out to remote agents. In this capacity as an Accepting proxy, the SCRouter is essentially performing a bet on behalf of the infrastructure at the local agent that a remote service can be found.

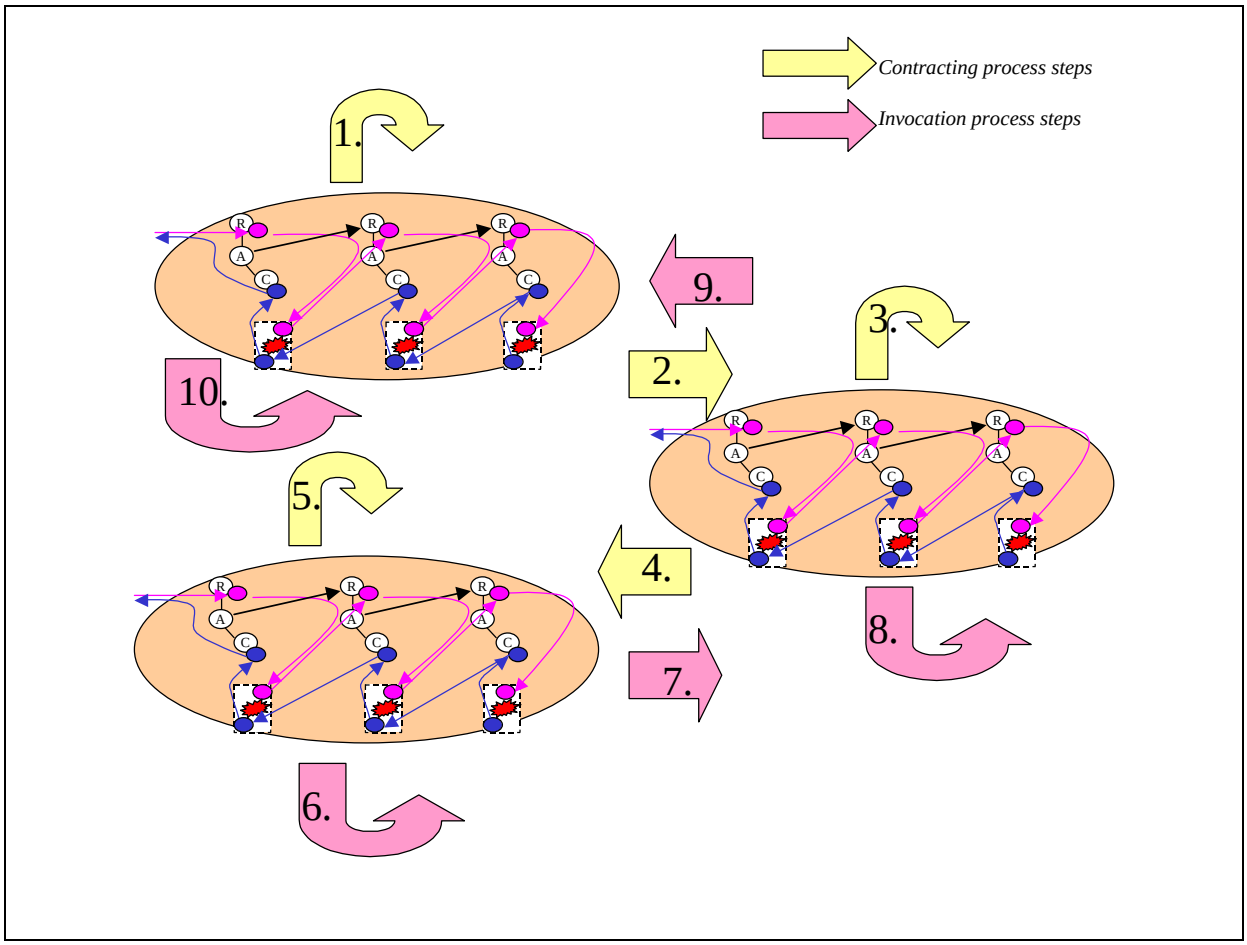

Figure 6 Illustrates the inter-relationship of "Contracting" vs. invocation in a distributed SC environment. Contracting process propagates "outward". Then after successful completion, invocation progates in the reverse direction.

At this point we'll briefly introduce an example and highlight several notable details about the SC interactions. The example itself is discussed more thoroughly in the accompanying software manuals.

Figure 7 illustrates the dynamic nature of SC workflow instantiations. The top two screen captures represent the state of the workflow early on in the assembly process – momentarily after an external Request has been injected into the agent EXAMPLE1. The visible cascading structure

Request->Accept->Request->Accept etc.

reflects the structure of the workflow within the EXAMPLE1 agent. This structure indicates that the EXAMPLE1 agent received commitments from two local services (TEST1, TEST2) and had gone off and was trying to find a dependent service (TEST3) elsewhere. Thus, at the end of this structure, there is a link (URL) representing the jump from the EXAMPLE1 agent to the EXAMPLE2 agent. These steps are represented as (2.) and (3.) in the schematic in the middle of the diagram.

The screen capture at the bottom of Figure 7 represents the stabilized workflow (time passes) from the perspective of EXAMPLE1 agent. You will note two other workflow structures; one represents the "switchback" (7.) in the schematic – the agent EXAMPLE1 contributes services at two very different points in the workflow life-cycle. The other workflow fragment represents the involvement (recruitment) of a Gauge service to satisfy a Contract Constraint evaluation.

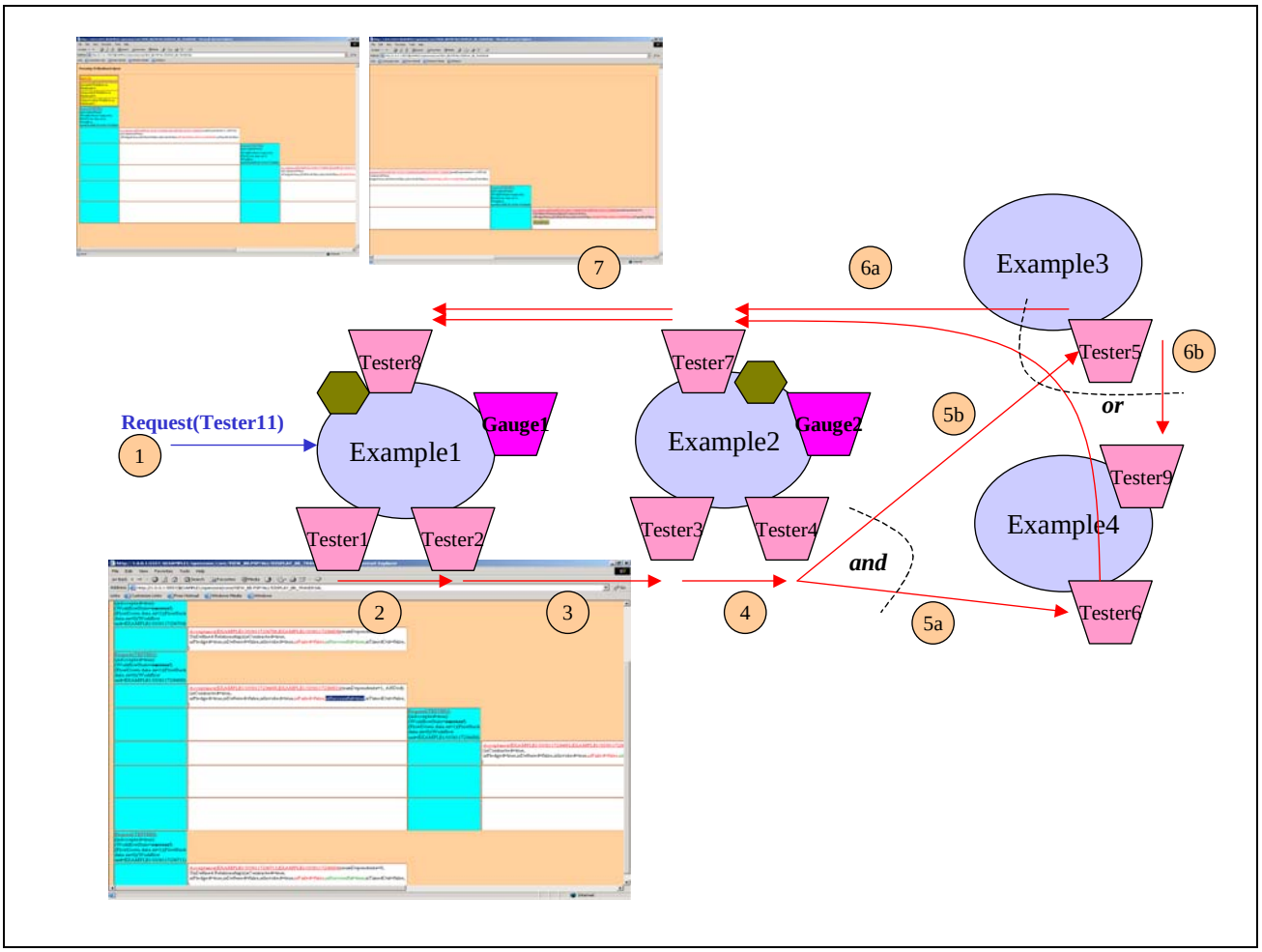

Figure 7 Time-phased view of the Blackboard of an example agent. Critically it illustrates the dynamic nature of the Blackboard as SC events drive formation of the workflow. This example is taken from the REGRESSION example described in the accompanying software reports.

A number of properties are visible in the user interfaces in Figure 7 that are worth highlighting to recap critical high-level SC ideas.

- **isContracted=boolean**, *Is there a Contract associated with Acceptance?*
- **isPledged=boolean**, *Has the Request been Accepted?*
- **isDeferred=boolean**, *Has a Request been accepted by an SCRouter (for remote exportation)?*

- **isInvoked=boolean**, *Has the Contract been invoked (Executor)?*
- **isFailed=boolean**, *Has the Contract been invoked and failed?*
- **isSuccessful=boolean**, *Has the Contract been invoked and successful?*
- i**sTimedOut**=boolean *Has there been an attempt to invoke the Contract but it timed out? Nuance difference for the case of Requests sent remotely – has Results come back yet?*

An interesting (and subtle) point worth highlighting here has to do with the earlier mention of a *Service Neighborhood*. In the example of Figure 7, the question posed to the EXAMPLE1 agent concerns how far to go afield to look for service TEST3. TEST3 is the service that the TEST2 service claimed as a dependency – a Request beneath its Acceptance. The simple reply is that there is some notional neighborhood that surrounds a particular service from within which candidates should come. The answer then is that the service should come from the neighborhood of the requester.

What this neighborhood denotes and its exact shape and size depend upon the application and the routing used. Thus, in the 2002 demonstrations, the neighborhood of an agent with respect to a particular service was defined as:

1. The set of agents with registered matching services in the Yellow Pages.
2. The set of agents with whom an agent has dealt with in the past for a service.
3. A preconfigured set of **N closest agents** (arbitrarily defined in the demonstration scenario).

This is not a general definition. Other SCRouters may choose to instantiate other algorithms and approaches. During demonstrations, 3 (above) is tuned by ourselves to throttle the performance of an SC system based on the speed of the machine(s), connectivity, the interest-level of the audience, etc. Make N in 3 too large and the cost of messaging becomes too pervasive.

All this poses an interesting question. *What distance may a service reach out? What is the measure of distance?* Is it the workflow graph distance, or some other measure of the separation within a process? In Figure 7 we see this point brought to home. We can see how agents can act as service middlemen intervening at different points in the workflow lifecycle. Thus the EXAMPLE1 agent can “ante up” services at three different points in the REGRESSION test workflow lifecycle:

- First providing initial services anchoring the workflow.
- Later providing supporting services to other services owned by other agents.
- Finally potentially a Gauge service in response to a runtime verification request (if a Constraint is issued against Gauge1 service).

3.6 Gauge Services

In the previous section we described how, from the SC infrastructure perspective, a Gauge can be many things. The infrastructure imposes no restriction on what a Gauge service provider can test; it only asks that it interact with the workflow in certain scalable ways. A Gauge must appear to the infrastructure as any other service, component or otherwise.

As with any other component service, a Gauge is "just another Plugin" (in the Cougaar sense), which can, in fact, act as a proxy for an external Gauge service (such as DASADA Runtime Gauge Infrastructure). This last point was a key feature that would have well supported a DASADA Phase 2 initiative.

Whatever constraint language an application uses to communicate with its Gauges, the definition of an SC constraint language is beyond the interest of the SC infrastructure. How an application speaks to its Gauges depends on its domain and the units of the measurement of its Gauges. It should be noted that a constraint language used by Gauges can be extremely simple. For example, in the 2001 and 2002 demonstrations, what was communicated to Gauges were an ordered set of threshold values. Or it can be, at the other extreme, a full language that is interpreted/executed within the Gauge services. In 2002 we were experimenting with more elaborate languages based on J-scheme [30]. In this case, an ASCII Scheme script was transported within the Constraint LDM object and was interpreted at the receiving Gauge. We convinced ourselves that this was practical within an SC system as currently defined.

For interoperability purposes, just as communities of related applications need to interact with common service ontology, they may need also common "languages" for describing constraints.

The following are a few sample "Gauges" - to illustrate the breadth of possibilities:

- A Gauge that tests the availability/version of a local Open Database Connectivity (ODBC) driver before use.
- A Gauge that tests via a Simple Network Management Protocol (SNMP) agent whether LAN connectivity can support intended application use.
- A Gauge that tests the current battery power level for the local node (preferences to services can be tailored to power levels).
- A Gauge that tests internet connectivity – e.g., test access to remote service before recruiting.
- A Gauge that tests system configuration - e.g., to insure that an application service may execute without conflict.

3.7 Hints (Directive)

Beside Constraints, another important type of Directive is the Hint. In the 2001 and 2002 demonstrations we illustrated the power of Hints by using them to drive the optimization of the workflow based on *roll-up times of past performance* and *individual service invocation* times.

Roll-up performance was measured with respect to the time it took the infrastructure to locate and connect services to satisfy Requests. These metrics were employed with respect to entire branches of a workflow, and performance metrics are indicators of the efficacy of the Service and Contract protocol and infrastructure. *Individual service invocation* times were employed with respect to the time it took to invoke a service. This measure was particularly useful for driving choices of substitute services. In this way otherwise identical services could be selected based on their different invocation latencies.

Based on these metrics (and other sorts are possible), we demonstrated how, through the use of only mechanisms local to individual agents, Hints can be computed and used to shape future workflows.

We were able to show:

- How Directives (Hints and Constraints) can be propagated along a distributed workflow to shape system "memory". Our approach was compatible with a number of reinforcement learning techniques. Our approach has a loose analogy with the human nervous system in that it flows information and integrates decisions along distributed workflow structures.
- How local adaptation can be driven by the "lateral inhibition" of substitute services using Gauges to optimize performance for large service fan-outs.
- How self-describing architecture descriptions can be generated to track dynamic adaptation. Descriptions were output in an Architecture Description Language (ADL) format for analysis and human comprehensibility.

We used "information decay" as a means to de-conflict generated performance expectations. Specifically, in the DASADA 2002 technology demonstrations we showed how "information decay" can be used to de-conflict Directives generated at various points in the distributed workflow. Essentially, the idea was as follows:

1. As performance metrics were "flowed back" along the workflow network they became "less convincing" to the infrastructure where encountered, as facts that the infrastructure, at the point of encounter, should use to base future decisions.
2. On the other hand, while less convincing, these facts had some value. Given enough confirming facts, the infrastructure at the point of encounter might choose to pay attention.

The rationale for (1.) is a question of relevance.

- Metrics farther away are best decided by agents (Hints) closer in. Recall that all agents generate Hints along the entire path that the metrics flow back.
- Metrics from farther away are against services farther away; the likelihood of intervening alternatives (substitute services, logical branchings) is greater.
- Metrics are less reliable (because of time separation).

The demonstration implementation worked as follows. Each piece of information within the infrastructure was represented by an Annotation object. Annotation objects were created by the infrastructure at various junctures and represent a sort of “message” from a named sender without an explicit receiver. The messages (Annotations) are inspected at various points in the infrastructure as they are propagated upstream with the Results. At each point the infrastructure may choose to exploit this information. For example, WorkflowAssessor Plugins (an SC infrastructure Plugin) uses Annotations from SCRouters to glean timing information about remote services.

Each Annotation has a value attached to it. That scalar represents the distance that Annotation has traveled at the point of inspection. Each time an Annotation crosses an agent boundary it is decremented by the infrastructure. Higher “decay” is translated into lower weights for Directives at the point where these Directives are created. So for example, the same WorkflowAssessor infrastructure Plugin creating Hints about timing expectations of remote services would interpret the distance that Annotation has traveled as how strongly it should “hint” about a particular remote service.

The motivation for using Annotations to compute decay is to de-conflict competing sources of information within the system. Recall that all agents are capable of generating Directives, hence the need for the consumer of Directives to be able to discriminate among those it receives. As Figure 8 suggests, some agents will tend to generate more Directives than others, but these will tend to be of “lower quality” as they are based on information that has traveled a greater distance (where distance = hops across agent boundaries).

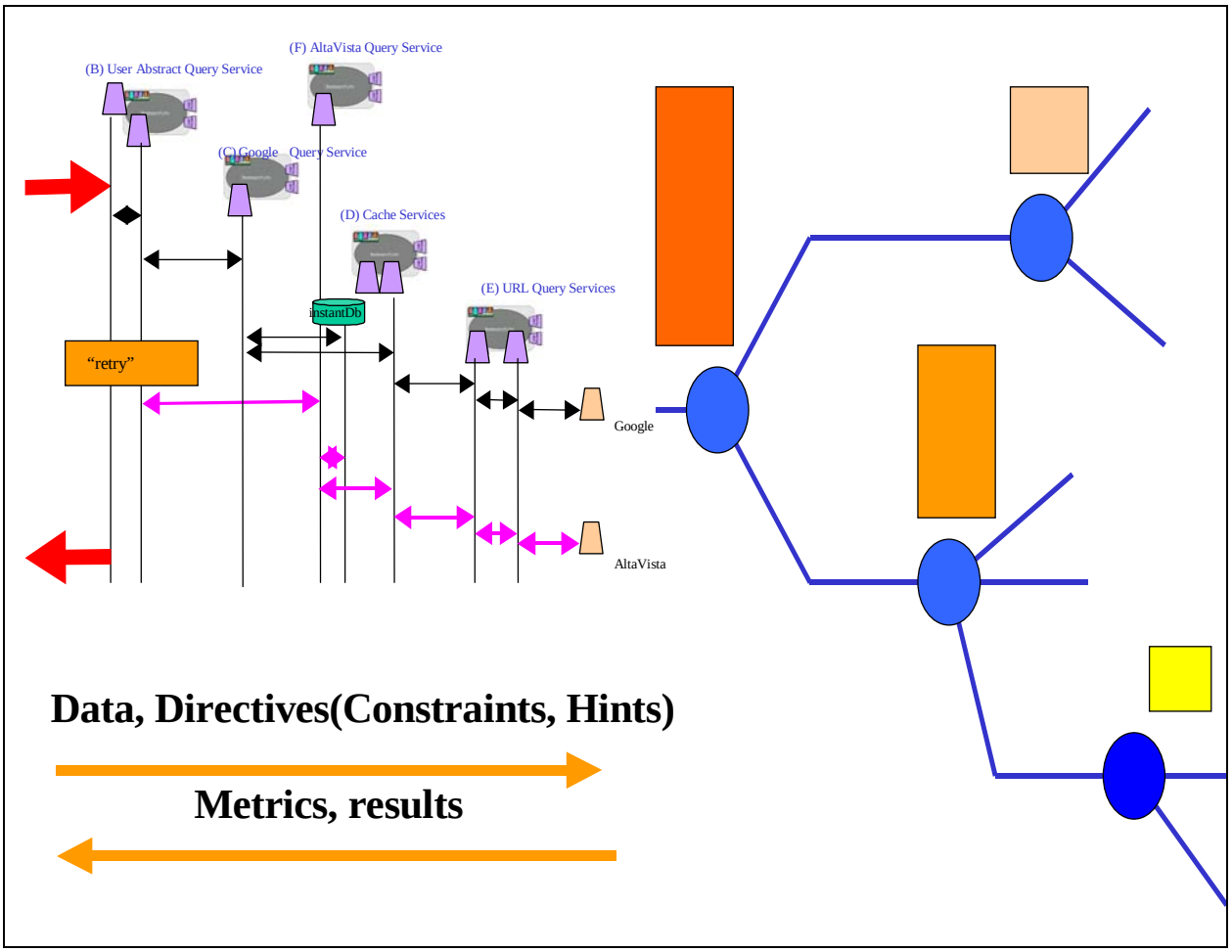

Figure 8 (Left side) Service and Contract workflows represent bi-directional information flows. (Right side) Directives (Hints and Constraints) are automatically generated and are weighted to reflect the quality of information consumed in their formulation. Upstream nodes tend to generate more Directives, but of lessor quality.

The SC Hints algorithm is based on the flow of information along the SC workflow network. The containers for information are instances of an LDM class called the Annotation. The Annotation is a text repository that may contain a number of individual attribute-value pairs of text information. The Annotation is a simple class that encapsulates text source to which the infrastructure might write as it flows past. Annotations are attached to Results – an LDM object that flows back with the execution results.

As a workflow is composed, Hints (along with Data and Constraints) flow outward. As the workflow is invoked (leaves first, working to roots), Results and Annotations flow inward. As Annotations flow inward, they can be inspected by the infrastructure at various junctures (private communication) to glean specific pieces of information. Annotations are collected and interpreted by the SC infrastructure at each agent to create Hints. Hints then are used to suggest how future workflow might be constructed to improve performance.

In the 2002 we demonstrated two types of Hints:

- "Bias SCRouter to send Requests of specific Concept to specific Agent" (Hint type = TYPE__DIRECT_ROUTING)

- "Bias Workflow Executor on Timeouts with respect to specific identified Services" (Hint type = TYPE__INVOCATION_TIMEOUT)

Hints come in two flavors: *Strong* and *Weak*. Whether a Hint is Strong or Weak is determined by the weight associated with the Hint. The current very simple algorithm is given as: each Hint type has a constant *Weight Threshold* associated with it. If the weight is above threshold, then they are Strong, else are Weak.

What it means for a Hint to be Strong or Weak is context-dependent upon the consumer of that Hint (an infrastructure component). So for example, a Strong Hint of type TYPE__DIRECT_ROUTING will be interpreted by the SCRouter to mean “route this Request to the target location at the exclusion of all other considerations”. The Weak version is taken by the SCRouter to be a suggestion that it may ignore in favor of other information it has access to – e.g., an external Yellow Page service. In the 2002 demonstration, the SCRouter treated Weak Hints probabilistically when it had alternative information such as from a Yellow Page service.

The intent was that these context-sensitive interpretations would be altered by Policies. In 2001 we demonstrated how Policy decisions could be used to tune whether or not Agents would ignore incoming Constraints.

3.8 Constraints (Directive)

Using the described SC building blocks, more sophisticated workflow behaviors can be achieved through specialization of the infrastructure components (for examples, see the Supporting Investigations section of this paper). Another means of *growing the workflow* is by extending the infrastructure with new capabilities (components) and through corresponding extension of the supporting SC protocol. A third area where significant customization is possible is by introducing new Directives for use by the SC infrastructure. One important type of Directive is the Constraint. In the 2001 and 2002 demonstrations, we illustrated the power of this idea via one type of Constaint - the *Contract Constraint*.

Contract Constraints are SC elements that are concerned with the assembly/execution of a workflow. Constraints can be bundled with the top-level service Request. In this case they would represent a requirement about how Requests are to be performed or interpreted. A Service Provider can also tag a Constraint to a Request that it Accepts. From whatever point a Constraint is introduced into an SC workflow, Constraints are propagated downstream from the point of their insertion into the workflow graph. Constraints, as do all Directives, propagate towards the leaves from where they were inserted. See Figure 9.

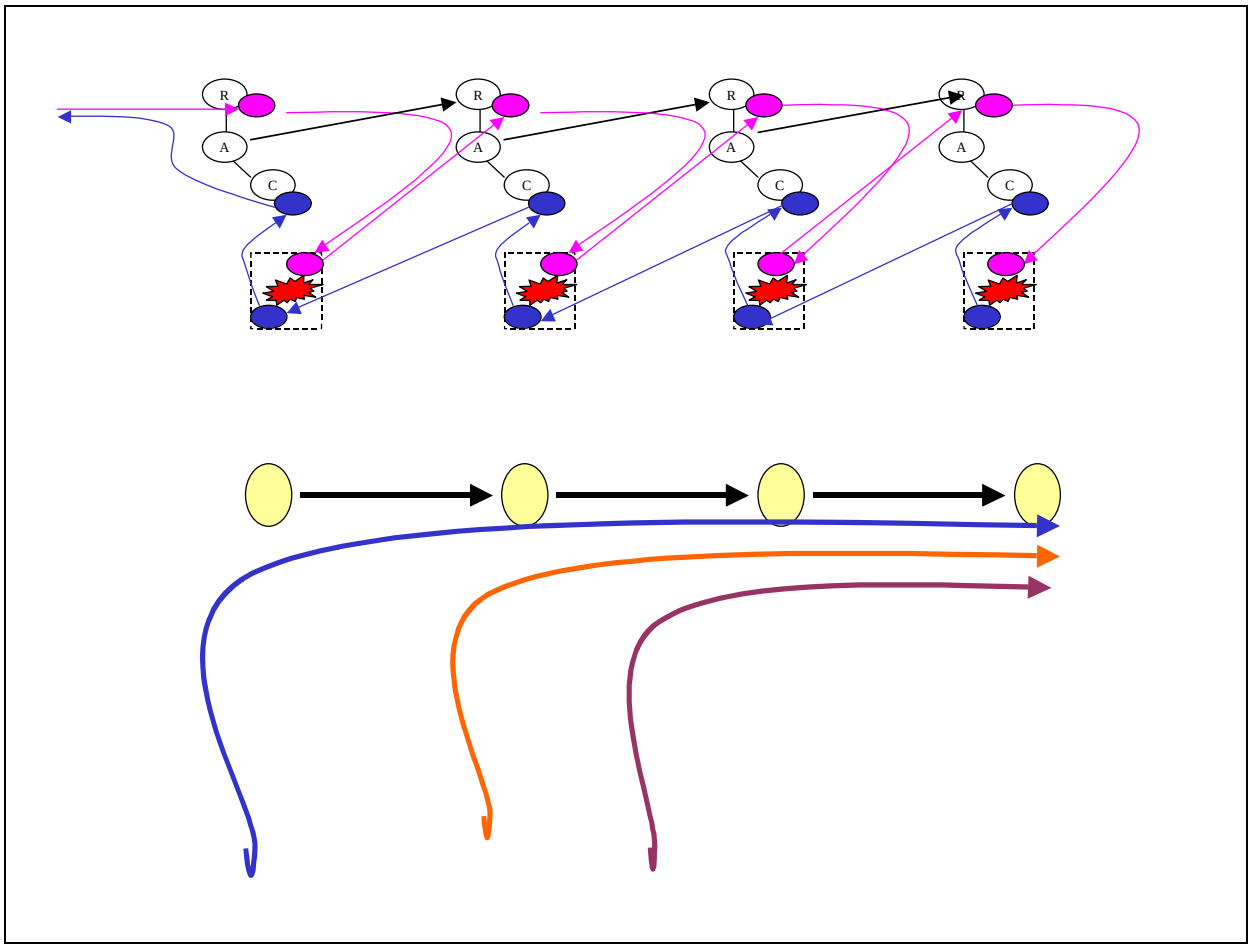

Figure 9 . Constraints (as well as other Directives) flow "downstream" from their point of insertion in an SC workflow. Hence, a Constraint inserted at a point in a Service Branch (Request) will "Govern" or influence the service branch beneath the point of insertion.

Contract Constraints govern the invocation of services. They identify the service signature of the relevant Service Providers. Services are identified by name in the name-space of the community of Service Providers within which the workflow spans.

Service and Contract Constraints specify a *target service*, a *Gauge service*, and an optional *Constraint expression (string)*. See the example below.

```
//

// deprecated Plan Service Provider "servlet" model for accessing Agent
Service (ConceptService)

//

conceptService =
(ConceptService)pd.getServiceBroker().getService(this,ConceptService.class,
null);
```

Constraints are attached to Requests. (In the current infrastructure, only “Contract Constraints” are used.) Contract Constraints are attached to a Request and govern all Contracts associated with any Acceptance “beneath” that Request.

Before service Contracts are invoked they are checked for any Constraints that govern them (the Constraint identifies the contracted service provider). For every Constraint found, another Gauge service needs to be recruited (via the same Request/Acceptance/Contract assembly paradigm) and invoked as a test.

Gauge services have access to the Contract data, to a constraint expression as part of the Constraint, and to the normal Service Provider and agent runtime, enabling them to determine whether to accept a Contract. In making such a determination, a Gauge service, for example, may evaluate the data, may consult an instrumentation substrate (DASADA RTI), or otherwise examine evidence in its operating or network environment. If any of these Gauge services fails to respond with a Boolean *True* – then the PRE condition test failed and the target Contract is failed before it is invoked.

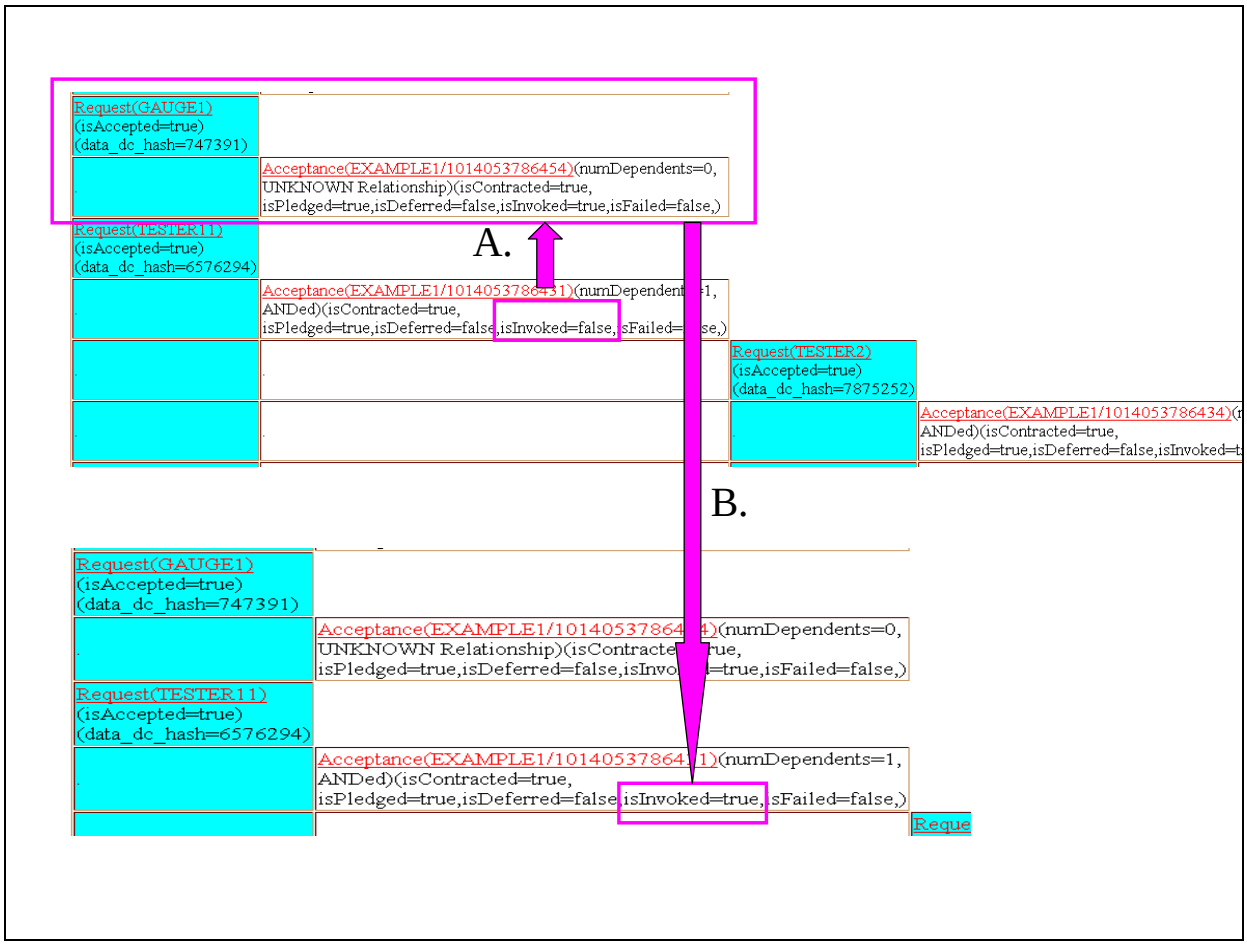

Figure 10: Using the SC Blackboard viewer – time-phased view of the runtime interaction of the PRE Contract Constraint and Gauge recruitment.

Figure 10 illustrates the time-phased relationship of Constraint and Gauge services using an SC Blackboard viewer. The Blackboard viewer is a simple HTML rendering of the contents of an agent Blackboard. It displays the fragment of a workflow (assuming a distributed workflow) located at that agent. The figure also illustrates the state change associated with a Contract for a particular service of type TESTER11.

In Figure 10, the top-half shows the Contract for TESTER11 as not invoked. This is due to the dependency (signified by A) of this Contract upon the successful evaluation by a Gauge service of type GAUGE1 on the input data (PRE condition test). This dependency was stipulated by a Contract Constraint.

The bottom-half of Figure 10 shows the successful invocation of the Contract for TESTER11. This was possible after the dependent Gauge completed its successful evaluation of the input data. This is indicated in the figure by B.

Note that in the interval of time between A and B, the Contract of TESTER11 is effectively blocked. This is because that service cannot be invoked until the Gauge reports back a successful result.

In the SC system, the infrastructure actor called the Executor will activate a new service Request branch if an unresolved (outstanding) Contract Constraint is detected. This evaluation happens at runtime. An advantage of this "late evaluation" of Contracts is that Gauges can be inserted into workflow assemblies as they are actually needed. This means that we can design our constraints to conditionally request evaluations of Gauges based on prior results.

The process for assembling Gauge services in response to a Contract Constraint requirement is identical to the process used for other component services. A Gauge service itself differs from other component services in that its "service" is to interpret a constraint expression and return some evaluation rendered by that Gauge.

There are three useful consequences of this approach:

- It simplifies design and implementation.
- It enables the modeling of Gauges just like any other Service Provider (component).
- Just as with any other Service Provider, Gauges may rely on support form other services: a Gauge, as a Service Provider, can request additional services.
- It suggests a useful approach of parsimony: other kinds of Constraints might be developed that can reuse existing SC mechanisms.

From the perspective of the SC infrastructure, the list of public facets of a Constraint is at this time limited. The main ones are:

- A Constraint is a type that indicates to the infrastructure that special handling is required. For example, an Executor looks at a Contract Constraint and decides whether to apply to a PRE or POST invocation step.

- A Constraint identifies a target service type. In the case of when a Constraint is a Contract Constraint, the target service type identifies those services whose Contracts are to be evaluated.

- A Constraint has an evaluation service type. This is the service that will be summoned by the infrastructure to conduct the evaluation. This service is a Service Provider (in the sense of the SC infrastructure) of a specialized type (a Gauge).

A Constraint can also carry a private payload to the Gauge. For example, the Constraint can carry a set of rules or configuration to be used by the Gauge in conducting its evaluation. For DASADA this was a means of allowing us down the road to integrate with other contractor constraint engines and the like.

Thus, a Gauge in an SC system can be defined as a service that measures something of its environment (system) and returns a "go/no-go" (Boolean) signal that the infrastructure uses to decide whether to proceed with a particular branch of the workflow. A Gauge service may be provided with application context - by passing in data, constraint expressions, thresholds, or other guidance that the Gauge service may choose to use to help it decide.

While a Gauge that has been recruited to test a Contract Constraint must at least evaluate Contracts and return a signal (Boolean), it can also attempt some remedy as a side-effect. So for example, a "proactive Gauge service" may solicit another service (as a dependency) to perform some remedial activity. For example, given this constraint: "when a contracted service opens a socket, validate that this host has network access before you try to invoke this service," suppose a Gauge that can test for this exists. It may try (as a side-effect) to recruit and re-launch a new modem service (etc.) if it notes a failure.

On innovation of the SC protocol and infrastructure design is how Gauges are handled. In an SC system, Gauges are just like other services (Service Providers). The only difference is that Gauges can be recruited dynamically by the infrastructure on an "as needed" basis. So, for example, an Executor about to invoke a Contract might notice that an outstanding Contract Constraint now applies and then goes off and recruits a Gauge and its dependency services to provide a required evaluation. Figure 11 illustrates one scenario. Noteworthy are these points:

- Gauges are modeled just like any other component: they are Service Providers.

- Gauges can be factored from their service dependencies. A Gauge is a Service Provider, and as such it can request other services for additional inputs.

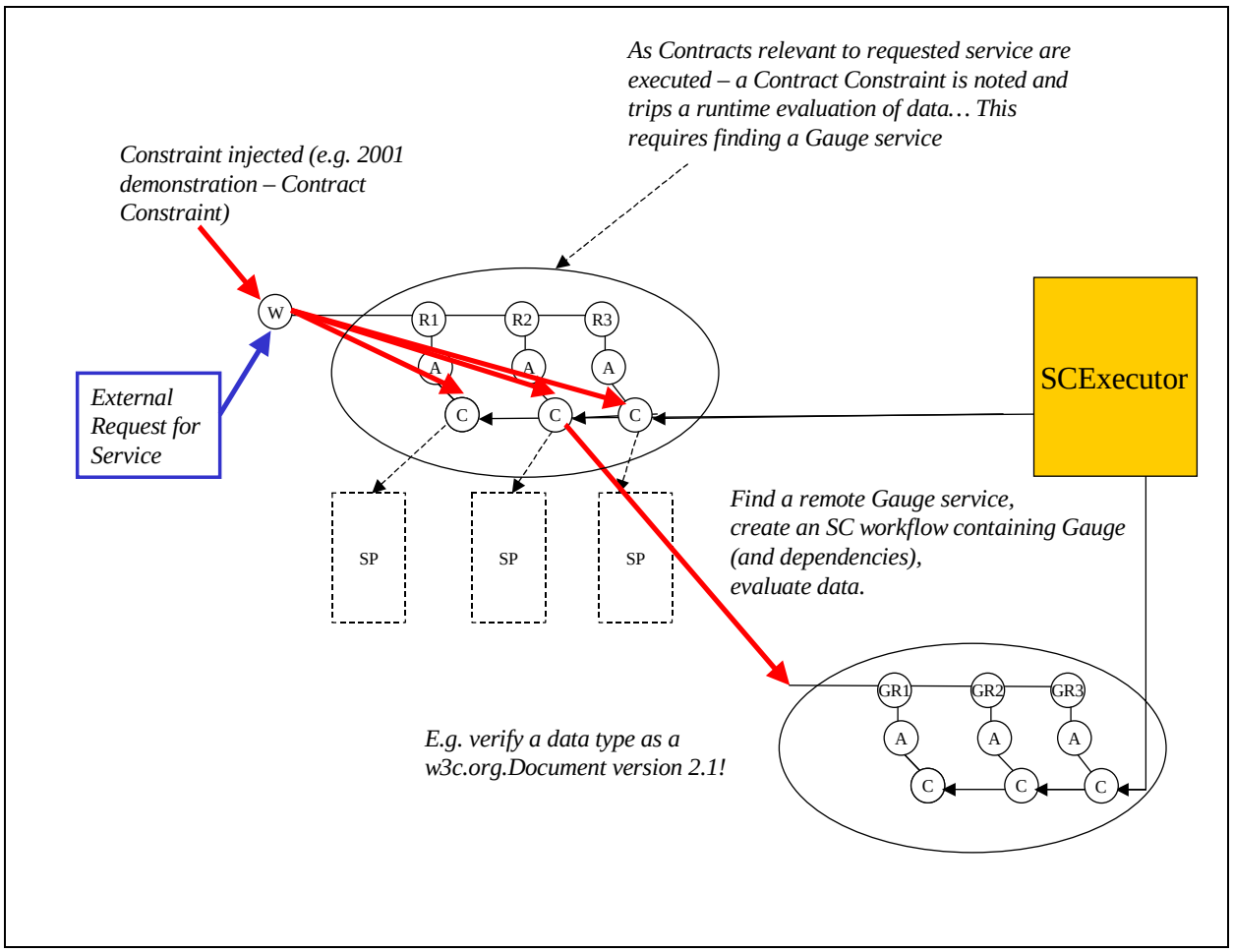

Figure 11 The Contract Constraint mechanism re-uses existing service Request/ Acceptance/ Contract pattern.

3.9 Policies

Service and Contract Policies are LDM classes whose instances are inserted into agents and used to modify infrastructure behavior at that location. In 2001 we demonstrated a simple version of this idea: that SC Policy objects could influence Agents to ignore and remove Constraints from workflows that pass through them. Policies, as they have been developed within the SC system, have been conceptualized to be *suggestive* – in that an agent may choose to ignore a Policy based on more compelling information it may be available locally. In the 2001 demonstration scenario (Figure 12), we illustrated how a Policy "switch" could alternatively induce workflows to fail and then to succeed. In Figure 11, a Policy would be inserted into agent (F.) that would strip off (and alternatively let remain) Constraints on workflows passing through the agent. In the demonstration we tagged workflows with impossible Constraints (that could never succeed) and through Policy changes, effectively turn on or off the connector (F.) to (D.).

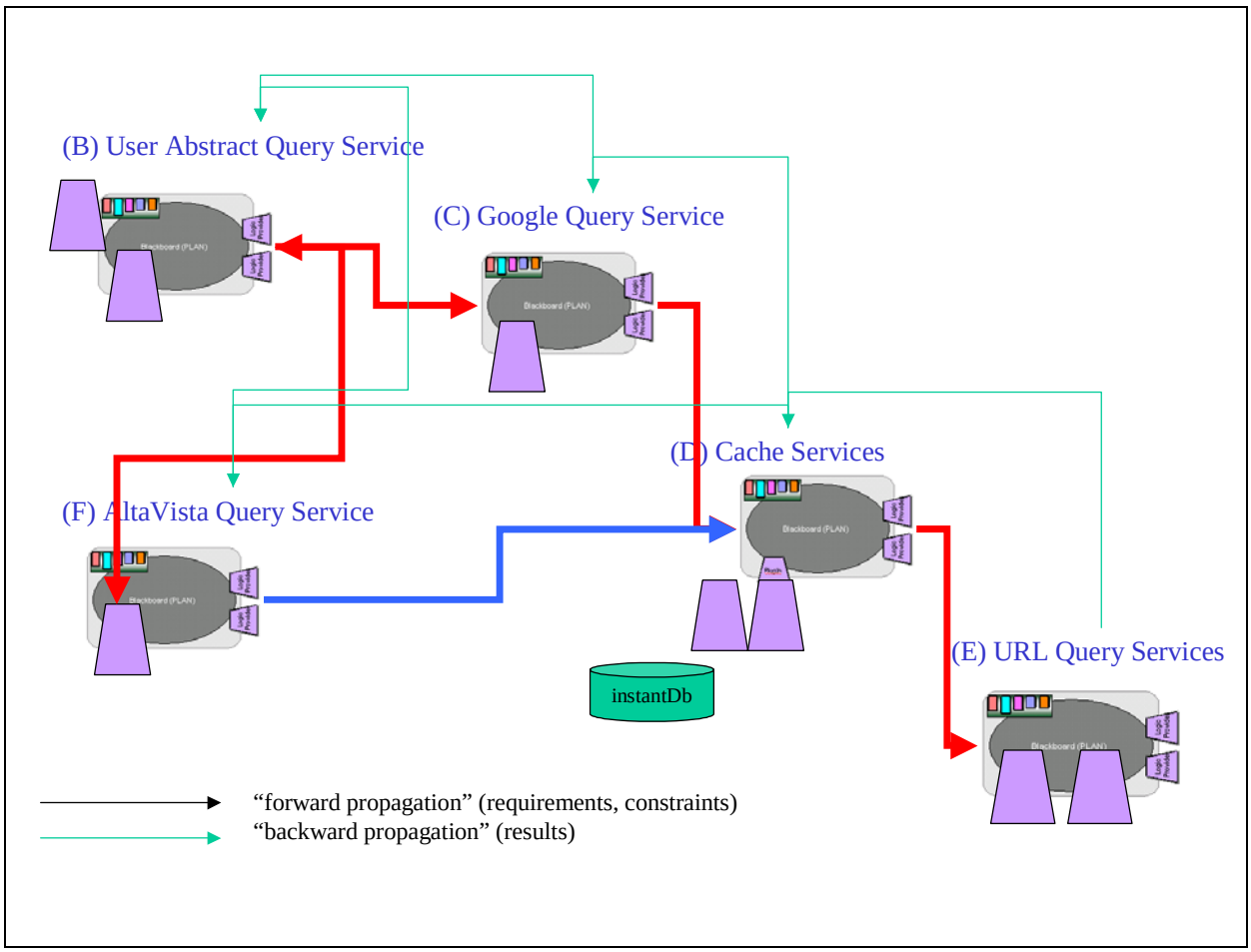

Figure 12: The 2001 DASADA demonstration featured use of Policies to turn-on and turn-off flow of Constraints.

3.10 XML

The Service and Contract workflow native representation consists of Java ™ objects located on the distributed agent Blackboard. These data elements retain the working knowledge (present and past) of its component workflows. We have experimented with a number of approaches based on XML Data Type Definition (DTD) and Schema-based techniques for specifying data translation. An early technique we used to obtain snapshots of the system was based on generating XML documents by serializing data objects on a local Blackboard. Contents (documents) across Blackboards were cross-linked using the native XML URL and XLink representations. Our subcontractor, JXML Inc., provided software and consulting services on techniques for manipulating XML models and parsing steps based on successive transformation steps of XML document data (reference Software Manual accompanying this report).

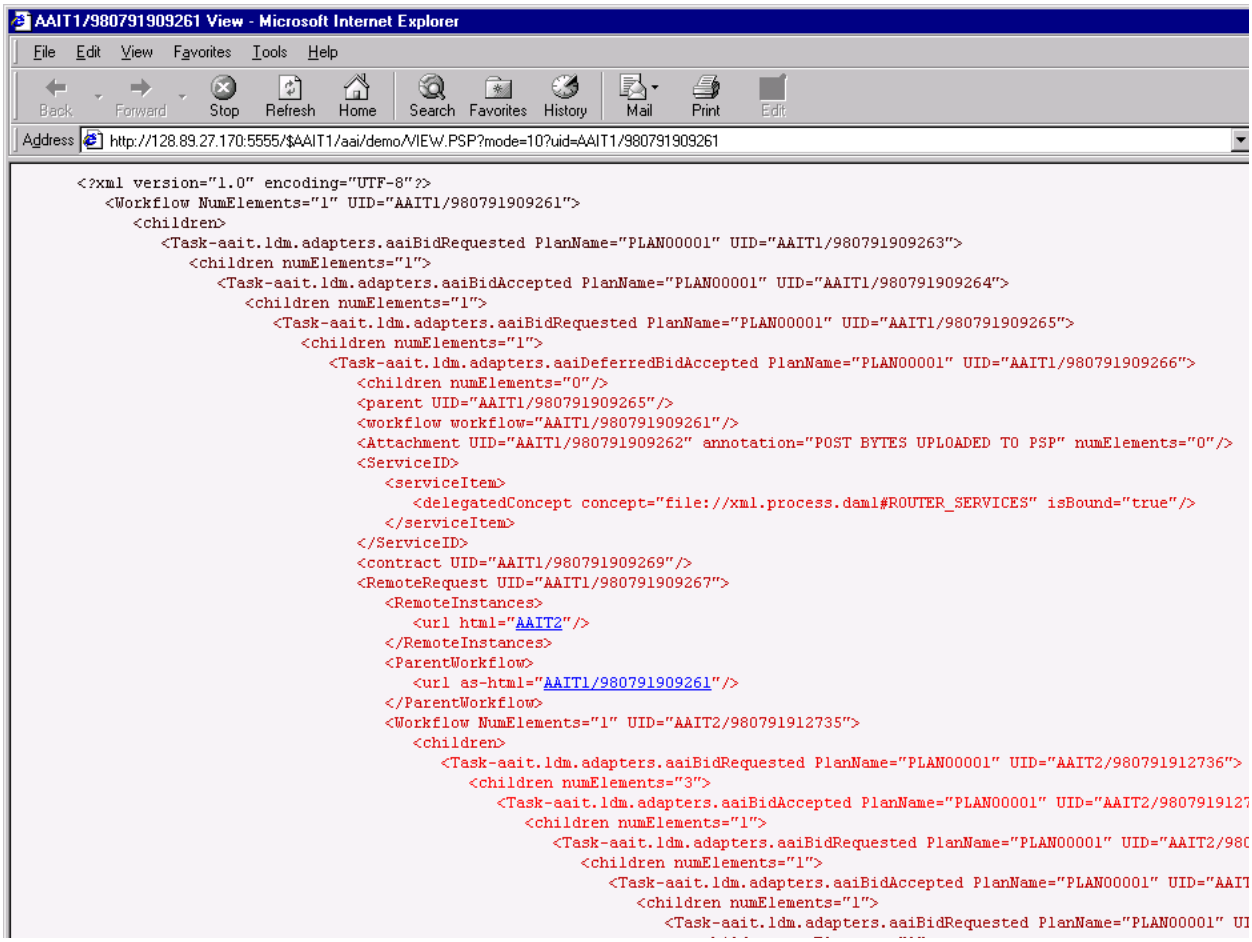

Figure 13. Single Service Chain in Workflow XML Serialized

XML serialization underlies much of the mechanism for translating Service and Contract workflow representations into and out of cleartext ADL (Architecture Description Language) and data document XML formats.

4. Demonstrations Overview

In 2001 we demonstrated the core Service and Contract (SC) ideas using a web services application (DASADA Technical Demonstration, Baltimore, [5]). We prototyped an *Abstract Query Engine* application. The Abstract Query Engine performed text-search, web-scraping, and database query services for ISI's *GeoWorlds* [7] Information Analyst tool. The 2001 GeoWorlds scenario involved a hypothetical Information Analyst using the Abstract Query Engine and the Geowold's client to analyze data from websources.

In 2002 we demonstrated a prototype *SmartChannels* application based on the AAI Toolkit (DASADA Technical Demonstration, Baltimore, [5]). SmartChannels provided a "fail-safe" capability to a GeoWorld's scenario by monitoring select critical connectors to remote services and intervening as needed. We embedded probes into the Geoworld's client software to detect problems. Upon detection of a problem, control and data would flow to the SmartChannels system. The SmartChannels system mirrored the failed connector and would route data to and from substitute services in lieu of the failed GeoWorld's services. For demonstration purposes, failures were induced.

In 2002 we showed how, using these building blocks, we can implement an adaptation model for distributed services based on Gauge feedback. In 2002 these building blocks were applied to a "Smart Connector" demonstration. A Smart Connector was an AAI Toolkit application that was able to shape its configuration and establish new connections to new (substitute) services should they be needed. New services are recruited in lieu of the old should performance constraints and expectations be violated.

Figure 14 below illustrates how our 2001 and 2002 testbed systems were constructed using the SC building blocks. In order for our systems to be credible from a testbed perspective, we sought sufficient breadth in our demonstrations to examine how the reactive and adaptation concerns of an SC workflow might interact within a complete system.

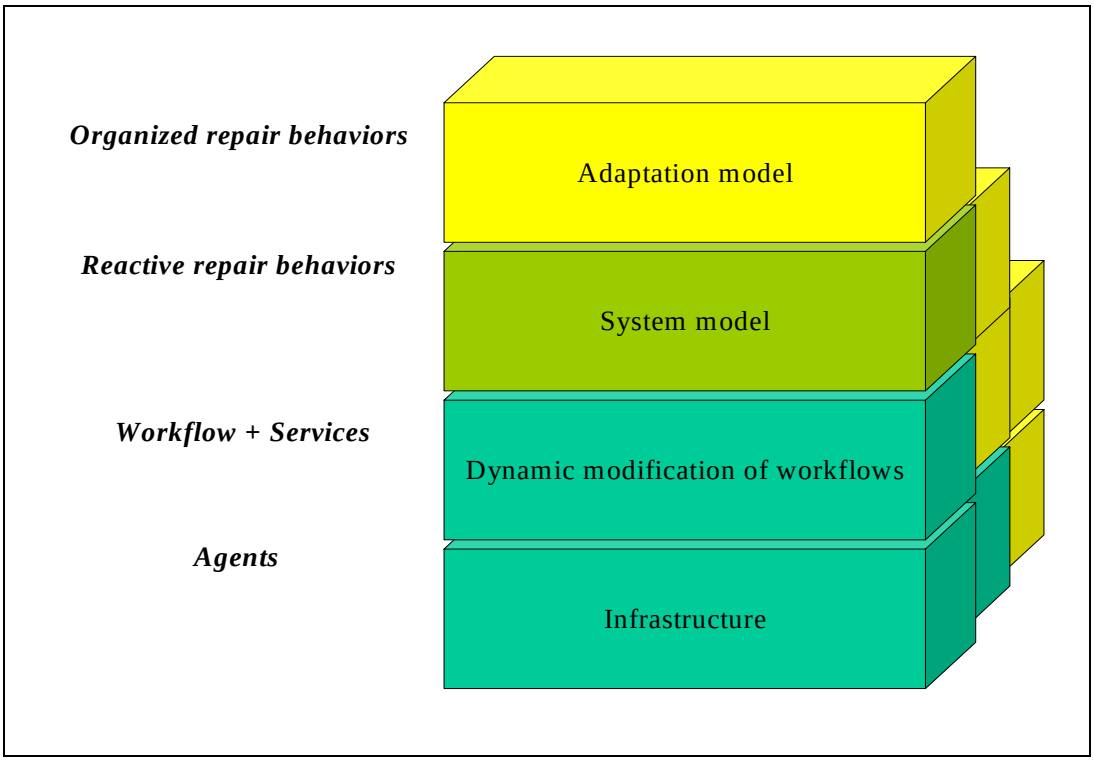

Figure 14 Testbed Layered Research and Technology Model

4.1 2001 Technology Demonstration

BBN-DASADA successfully developed and demonstrated a 2001 prototype of an SC application: The ***Abstract Query Engine (AQE).*** The AQE was used by a GeoWorld's client application (ISI) to query and obtain content from online search engines. The Abstract Query Engine was used in conjunction with other DASADA products to demonstrate an Information Analyst scenario at Pacific Command (PACOM).

The Abstract Query Engine networked almost two dozen service types (infrastructure and domain/application services) distributed on five nodes, to demonstrate a distributed "meta-search engine". Figure 15 outlines the basic demonstration scenario. The Abstract Query is ultimately reconciled with either a query to a Search Engine Service Provider. Search Engine Service Providers came in two flavors in this demonstration: Google and AltaVista. Each flavor of Service Provider knows what it had to do to issue a query and obtain usable results from their respective WWW search engine. For example, they knew (or rather knew the appropriate other services that were recruited) how to use database, socket management and text parsing and data aggregation services to return data that could be handed to GeoWorlds or Excel for display.

The demonstration illustrated how the SC mechanism can assembly the appropriate services into a distributed workflow that can satisfy the end-user's needs. Within the Abstract Query Engine system of agents, there is no initial pre-configuration. Services are wired up on-the-fly, as needed. This allows the system to respond to failure by reorganizing its configuration. In the 2001 demonstration, we induced failure of the first choice (Google) by insisting on an impossible Constraint. Once this occurred, the SC system would reconfigure itself around the AltaVista

Service Provider and its dependent services. Figure 15 roughly illustrates the configurations of the agents to satisfy the 2001 demonstration.

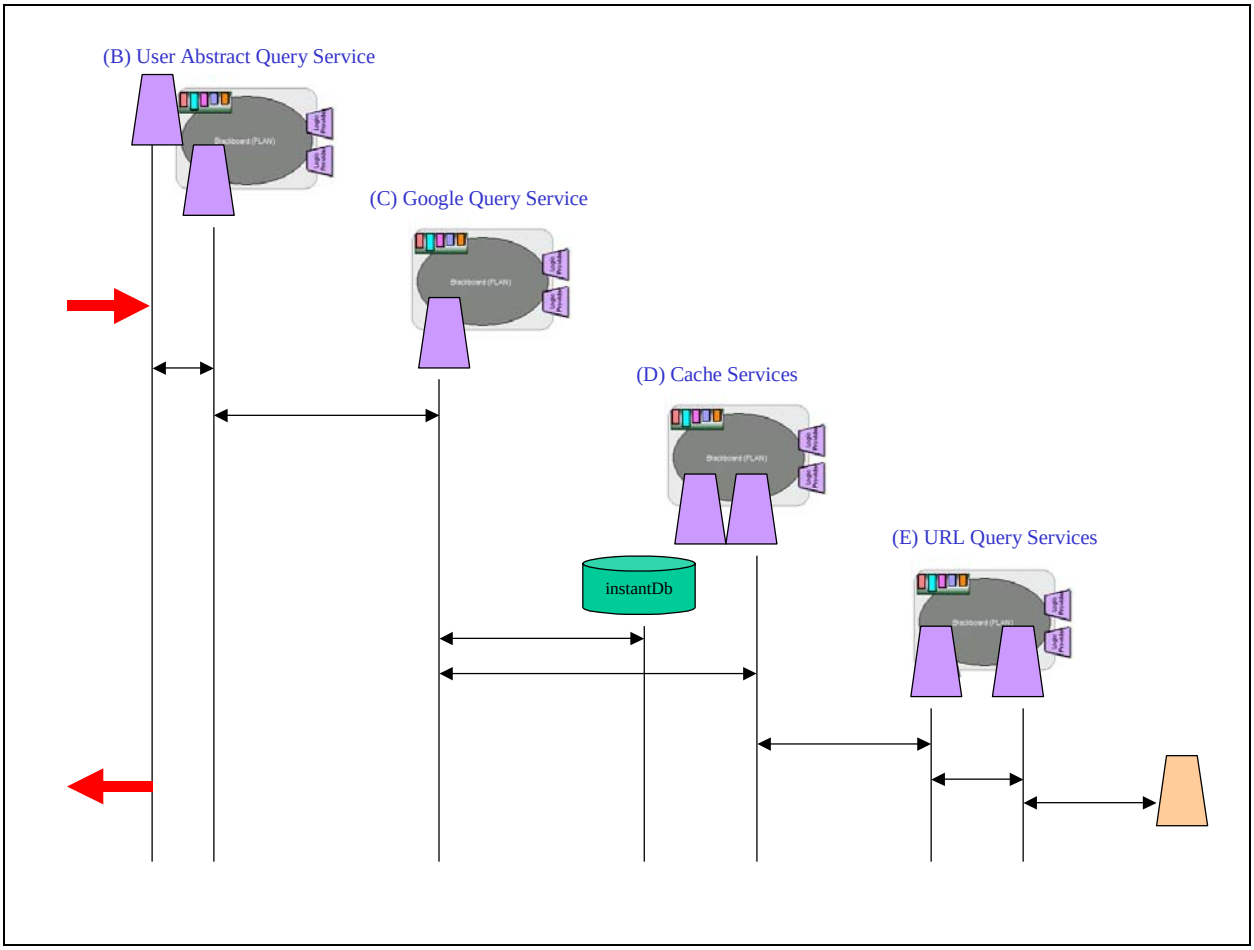

Figure 15 The basic scenario illustrated in 2001.

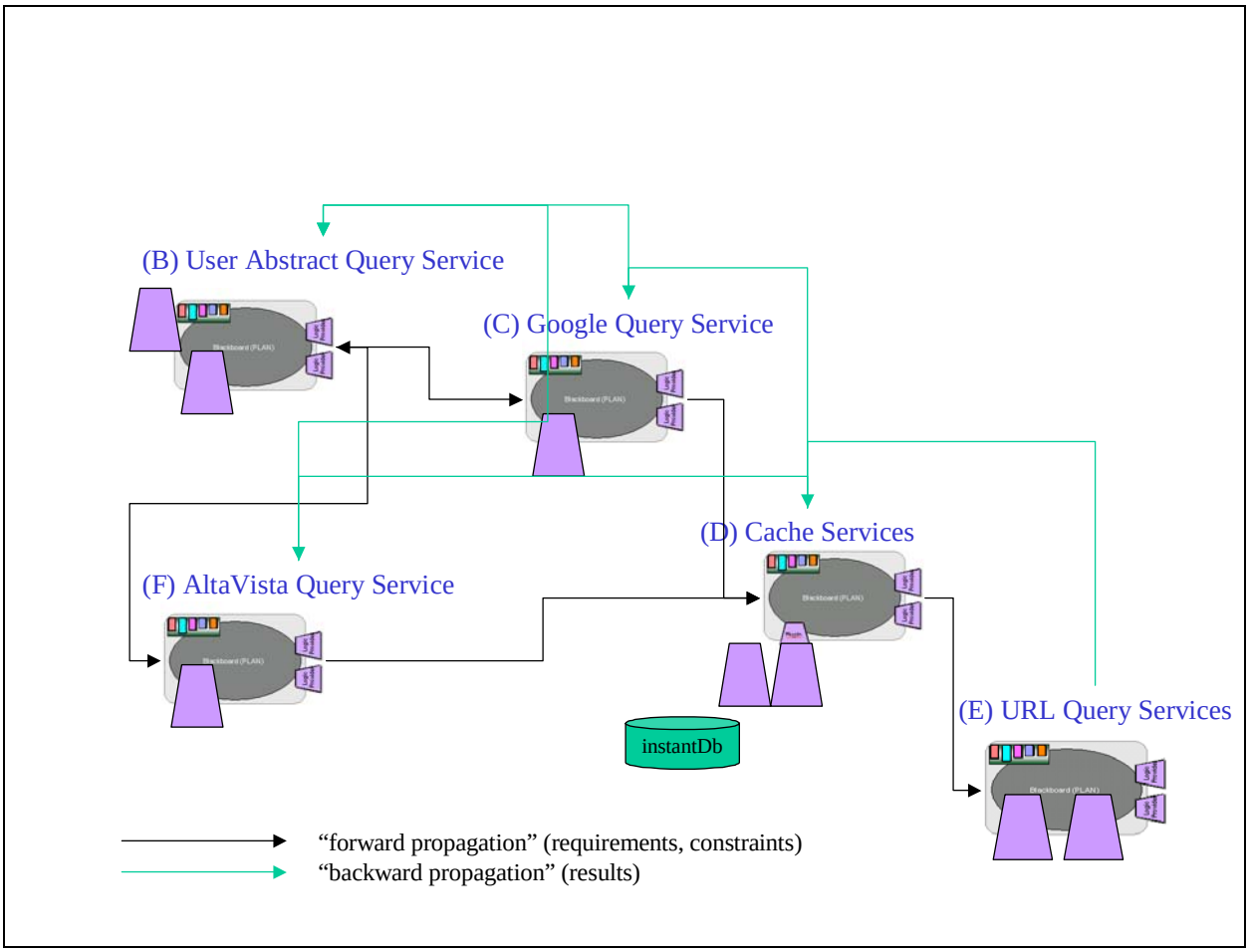

Figure 16 Abstract Query Engine (2001 Demonstration system).

Figure 16 describes the relationship of the Abstract Query Engine with the SC, AAI Toolkit, and Cougaar technology layers. It articulates the distinction between the application layers (AQE – 2001, SmartChannels – 2002, described later) and the SC and Cougaar infrastructure layers.

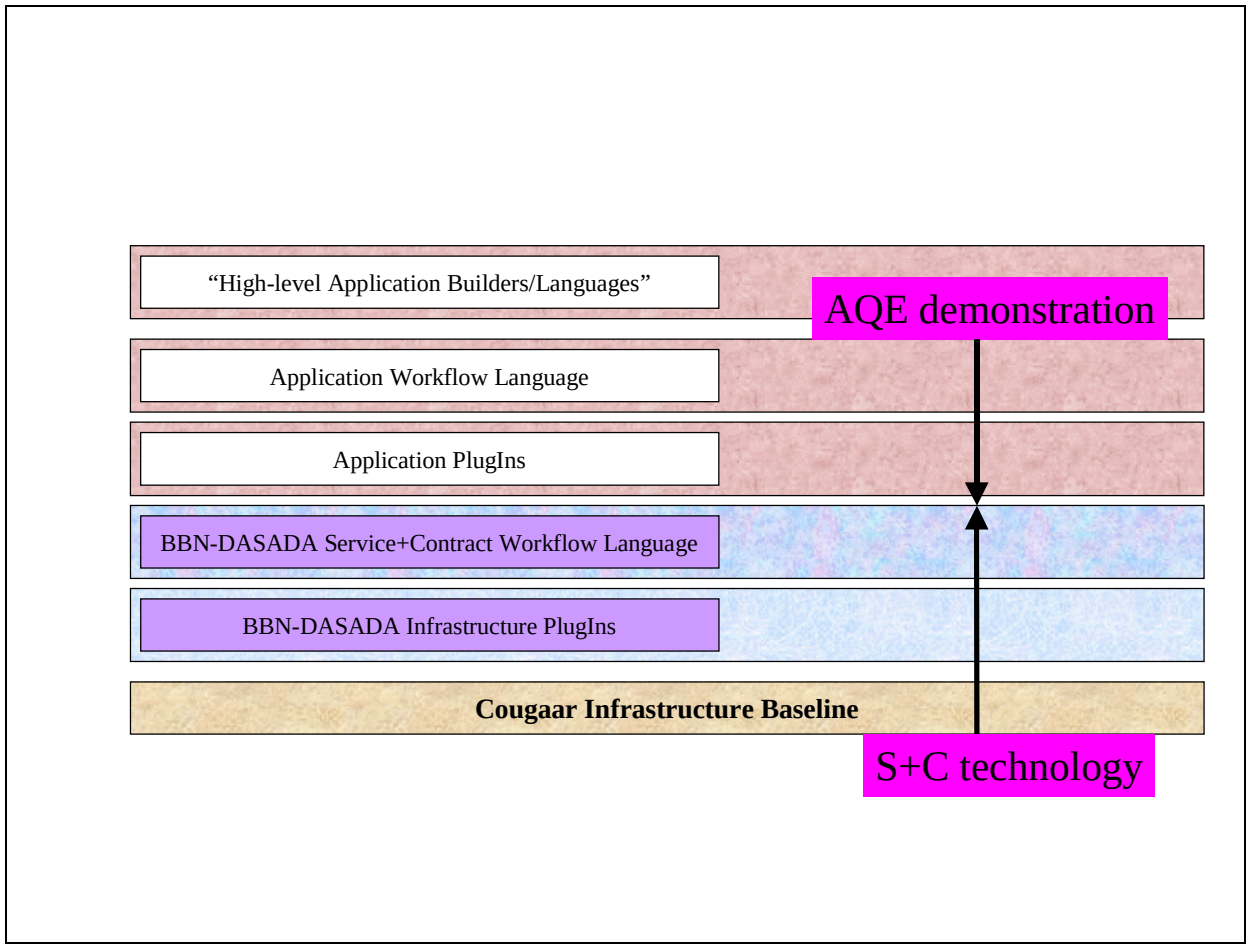

Figure 17 Diagram illustrating "layered" technology approach of BBN-DASADA. It is important to note that the main BBN-DASADA contribution ("SC" technology) is a not in itself an application but a platform upon which application workflows can be built.

Two classes of user interfaces were demonstrated in 2001 (Figure 18 and Figure 19) desktop application interfaces and browser interfaces. In Figure 18, shows that desktop applications such as ISI's GeoWorlds and Microsoft's Excel were integrated with the AAI Toolkit agent system. In the second case, HTML browser interfaces were used to show-off a variety of developer UIs.

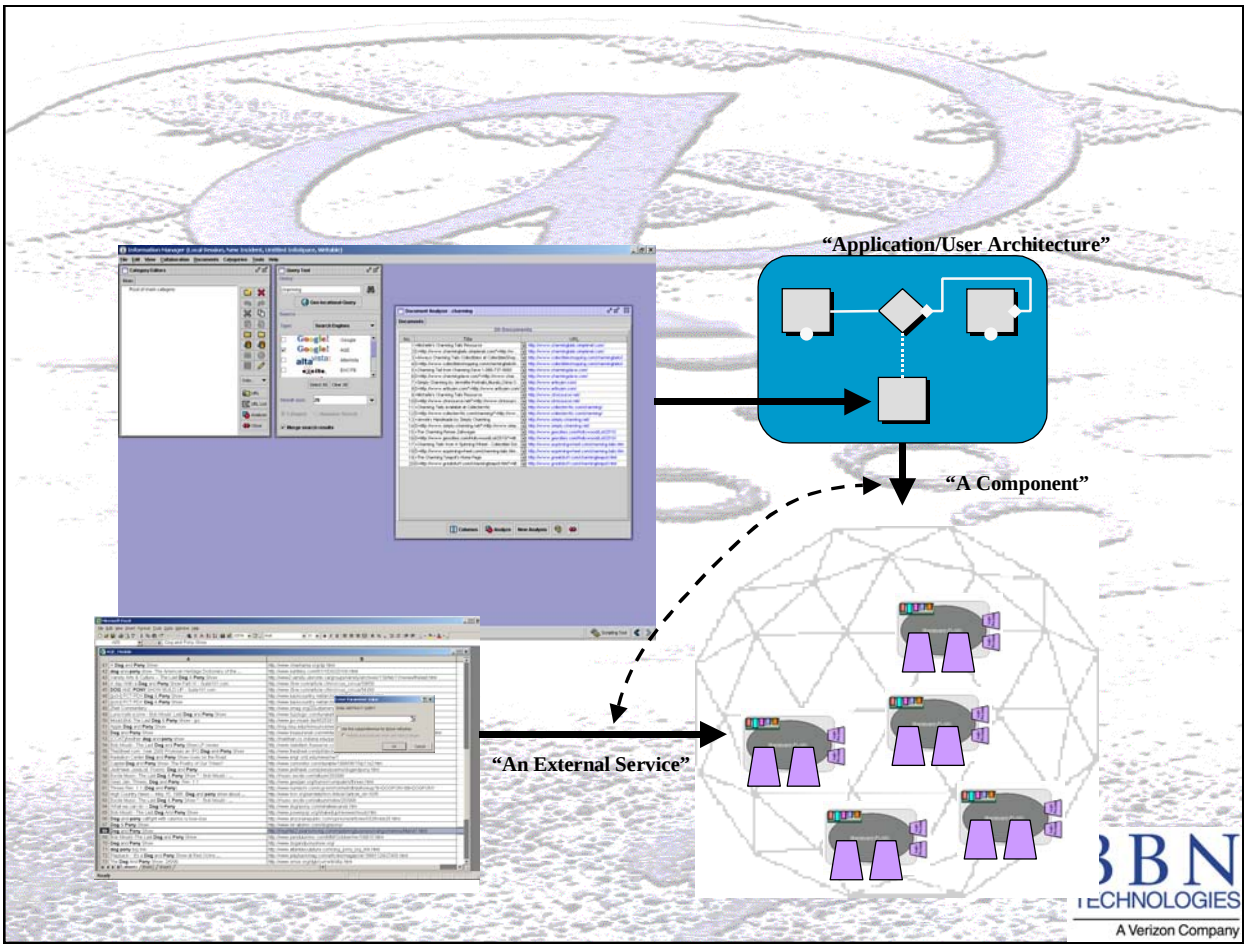

Figure 18 Abstract Query Engine (2001 demonstration) had two application interfaces: GeoWorlds, Excel.

Figure 19. Number of developer user interfaces based on the Cougaar webserver have been developed.

In 2001 we used a commercial hyperbolic tool [29] to view the interactions of the Service and Contract components during execution. Figure 21 shows these views in greater detail. Later, in 2002, we transitions to views based on an Architecture Description Language (ADL) representation. Hyperbolic visualization provided a useful paradigm for packaging and navigating a vast quantity of low-level SC events (publish/subscribe SC LDM objects onto Blackboards). However, it quickly became apparent that with a system of any size, a more abstract representation of architecture would be necessary. The events were captured and transported over an Apache Log4j logger channel (distributed) [31]. Figure 20 illustrates the process used in 2001 for visualizing Service and Contract events.

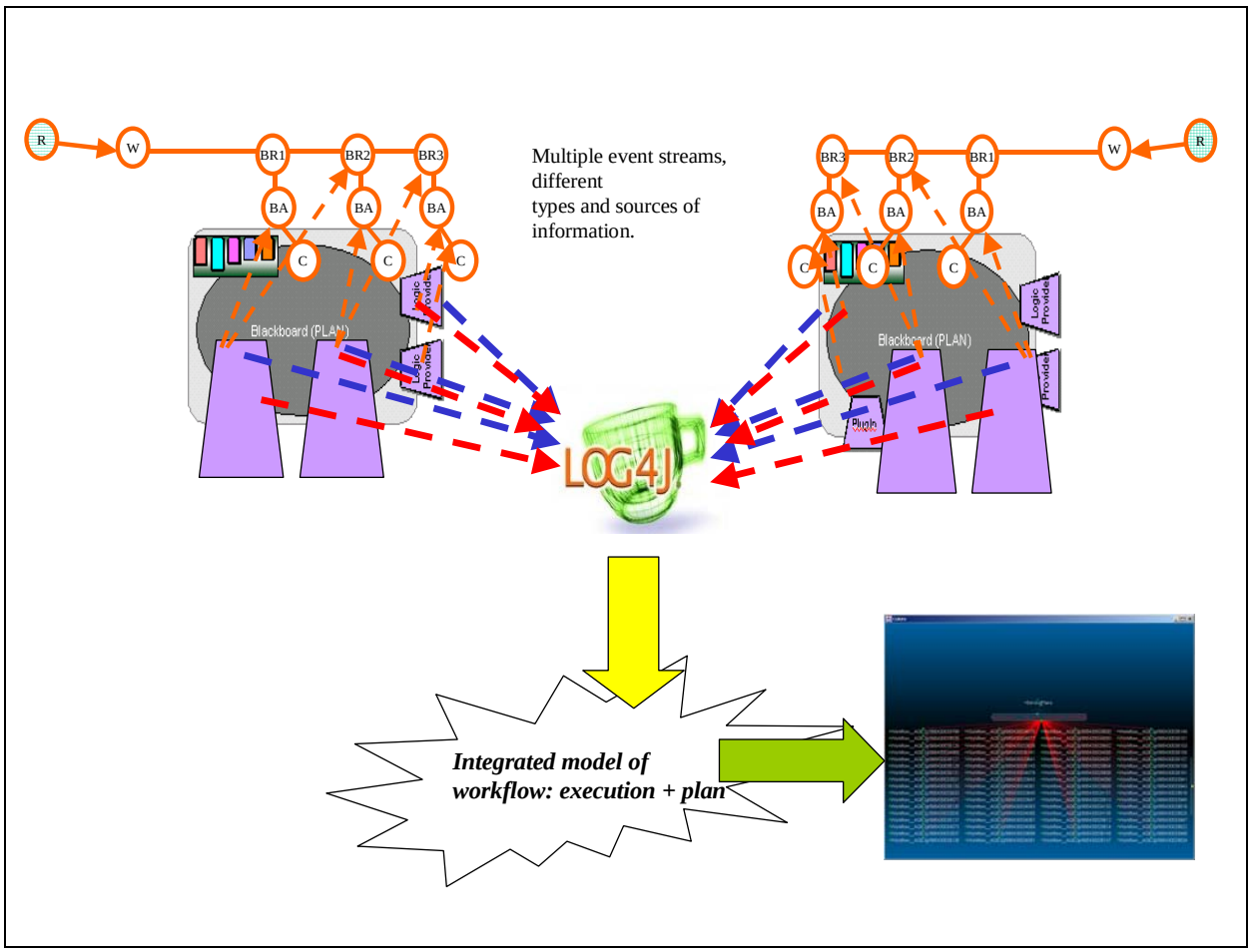

Figure 20 In 2001 workflow events are collected via Log4J.

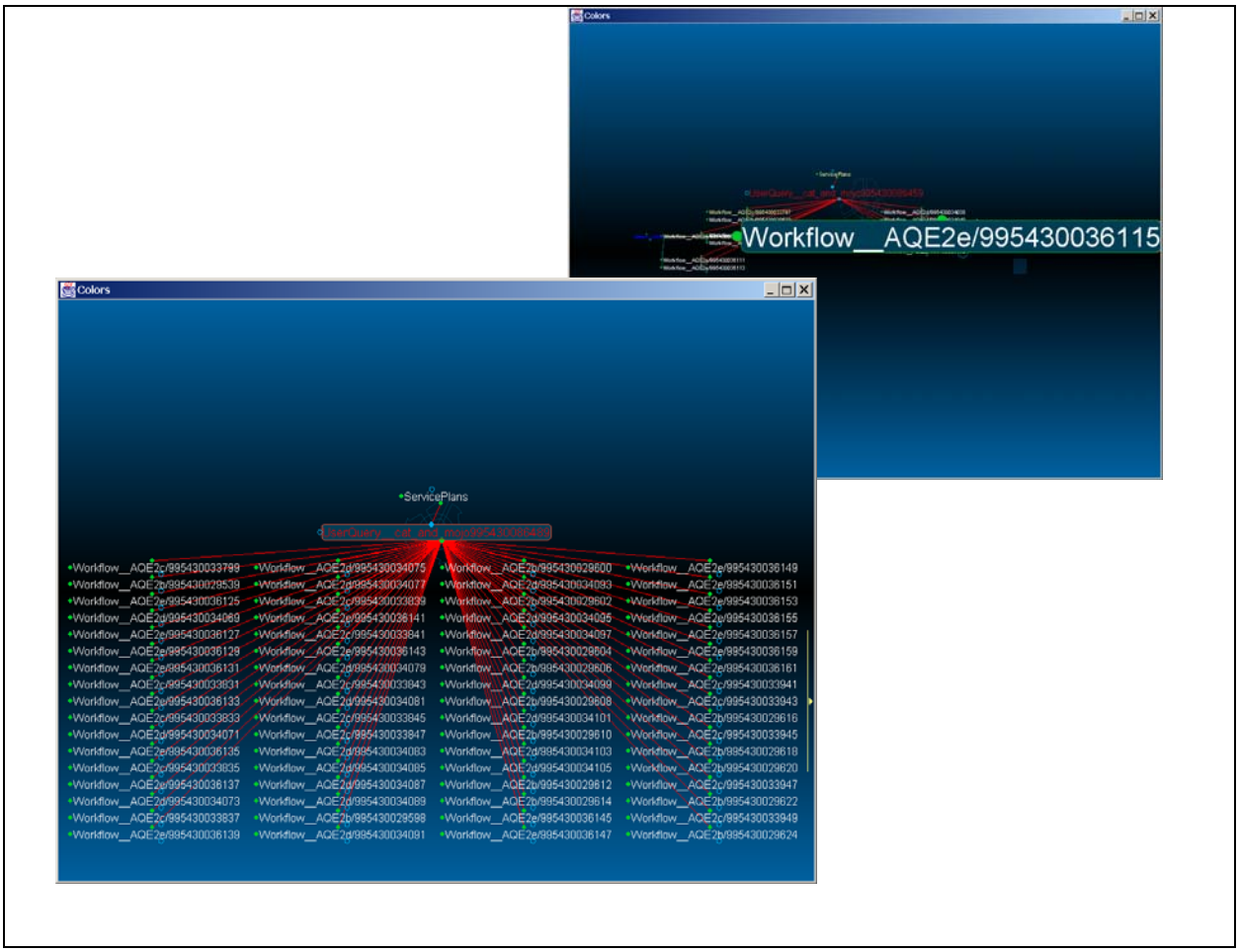

Figure 21 Constructing an architectural model for visualization from events.

Also apparent from the 2001 demonstration was the important role an instrumentation and feedback channel outside but integrated with the SC system (e.g., [3]) could play in enhancing the reliability of the system. In 2001, the Abstract Query Engine demonstrated the performance detriment (time/latency) of an SC system operating with a service in a failure mode without a Probe/Gauge layer to consult. The SC system continued to operate but was making decisions based on Policy assumptions about failure versus acting on actual knowledge. For example, when a downstream service failed catastrophically, the upstream agent had no means of discovering this. In the 2001 demonstration, it would decide failure of downstream services on the basis of “timeouts” (a Policy determined time interval).

Ideally, we would prefer an error signal (explicit message, or implicit missed health beat) to indicate when services enter troubled or failure modes. Of course this is not fool-proof – who monitors the monitor, etc.? However, a solution which integrates monitoring feedback organically into the distributed decision making process will be better off. In our paradigm, monitoring feedback from external infrastructure is integrated as another Gauge service. We provided limited demonstration of this use of the Gauge concept in 2002 (see CDIGauge in later section).

The failure scenario in the 2001 demonstration centered on the cascading cost of a lack of timely error signals in a distributed system. In our case, we illustrated how a single failure timeout can lead to a compounding of timeouts across a system. This implies a significant performance hit (time) as it's rolled up to the workflow root.

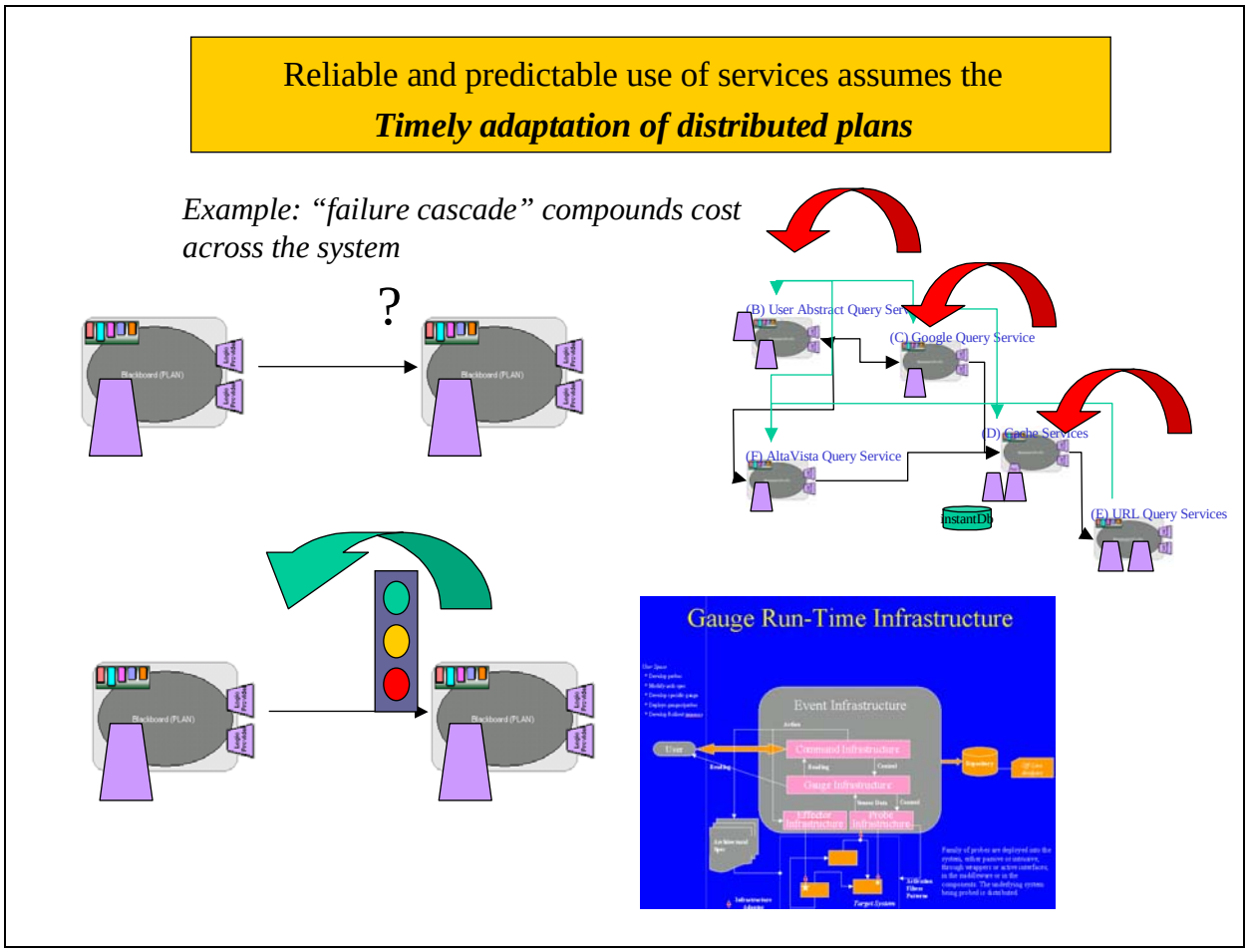

Figure 22 We demonstrated in 2001 the important role of a Gauge/Probe infrastructure (DASADA) in an adaptive system.

4.2 2002 Technology Demonstration

In 2002 we demonstrated an application of the AAI Toolkit called *SmartChannels*. This was a connector for insertion into SOS (System of Systems) applications that improved connector reliability. As in our 2001 demonstration, the scenario focus was ISI's Geoworld client application. We demonstrated how, using SmartChannels, one could improve the reliability of distributed systems such as GeoWorlds.

In the 2002 demonstration we highlighted capabilities supporting self-repair and monitoring in a distributed system-of-systems architecture. Essentially we created alternative pathways that "mirrored" the vulnerable connectors within the architecture and using agents could find and recruit new substitute service providers and gauges (including all their service dependencies) based on need. In addition to responding to faults, these agents learned how to improve their

performance by being more selective about which services they used. They used external guidance (Constraints) and their own previous experience (Hints) to make choices.

The reactive repair model is based on the intrinsic qualities of an SC (workflow) network. An SC network is able to dynamically re-configure itself based on the performance of constituent services. It can on-the-fly bring to bear Gauges to test operational parameters of the system and of services. It can seek alternative pathways and service providers based on need. These qualities are basic to the behavior of an SC network.

The gist of the adaptation model (Organized Repair in Figure 14) was to use flow of information amongst SC networks. In the forward direction, Hints are used to shape which services are used by the agents. Services may be selected subject to performance expectations (e.g., previous experience). In the reverse direction, metrics flow back and serve as the information basis for the infrastructure, generating new Hints for future use by the agents.

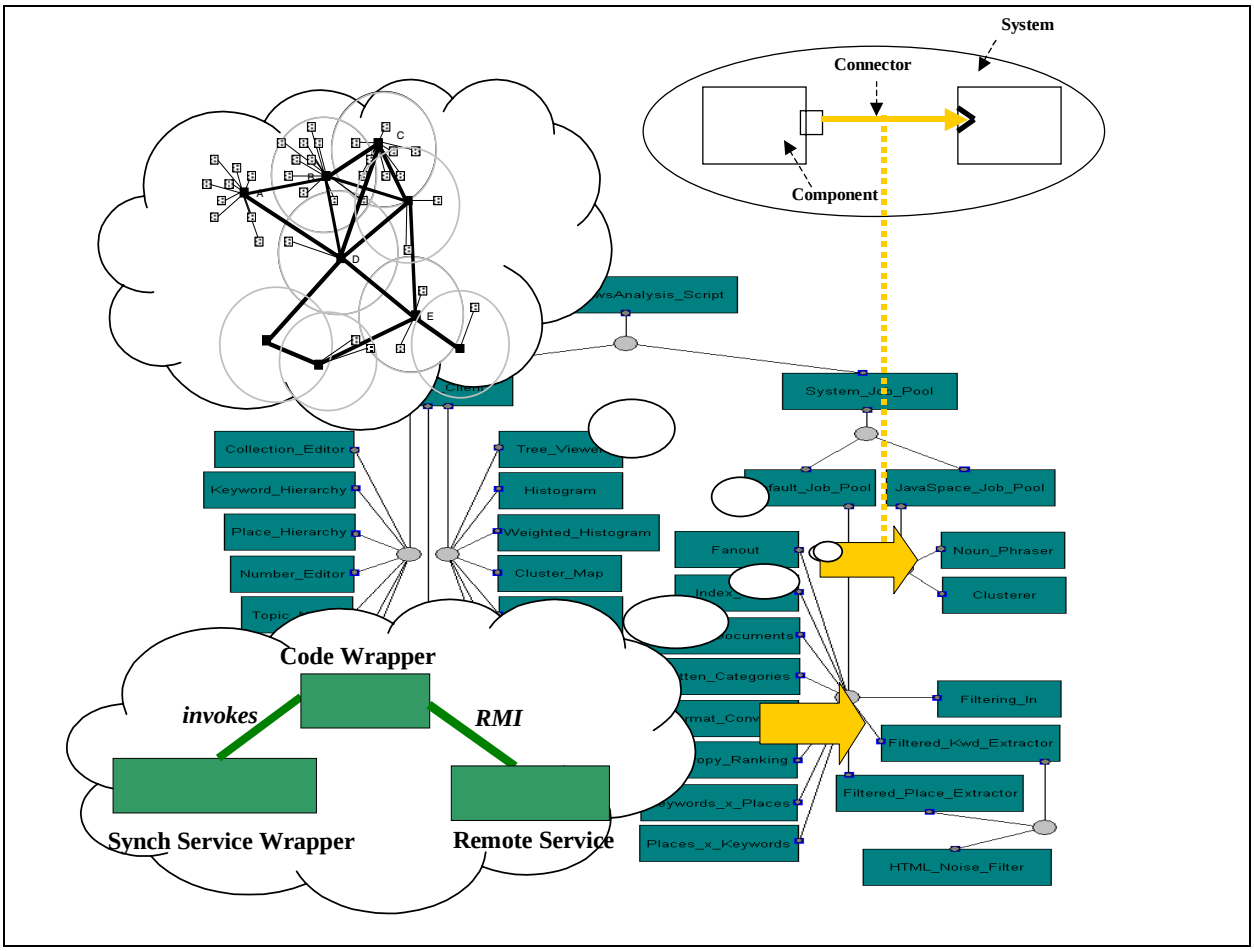

Figure 23 Architecture Adapter (Connector) abstracts internal details

Figure 23 is a schema of the 2002 DASADA technical demonstration (SmartChannels), where select connections between services in an intelligence analyst's application were mirrored using

an agent-based pathway. (Logically, the connection is depicted as a single connector, when in fact, in both cases, this is a simplification of the actual implementation.)

The SmartChannels system could circumvent vulnerable Geoworld connectors (e.g., JINI [11], RMI) via an alternative SC agent system. The SmartChannels system would reconfigure itself to reflect performance metrics. Figure 24 represents the idealized picture of the demonstration system. The actual demonstration system scenario threads appear later in Figure 27.

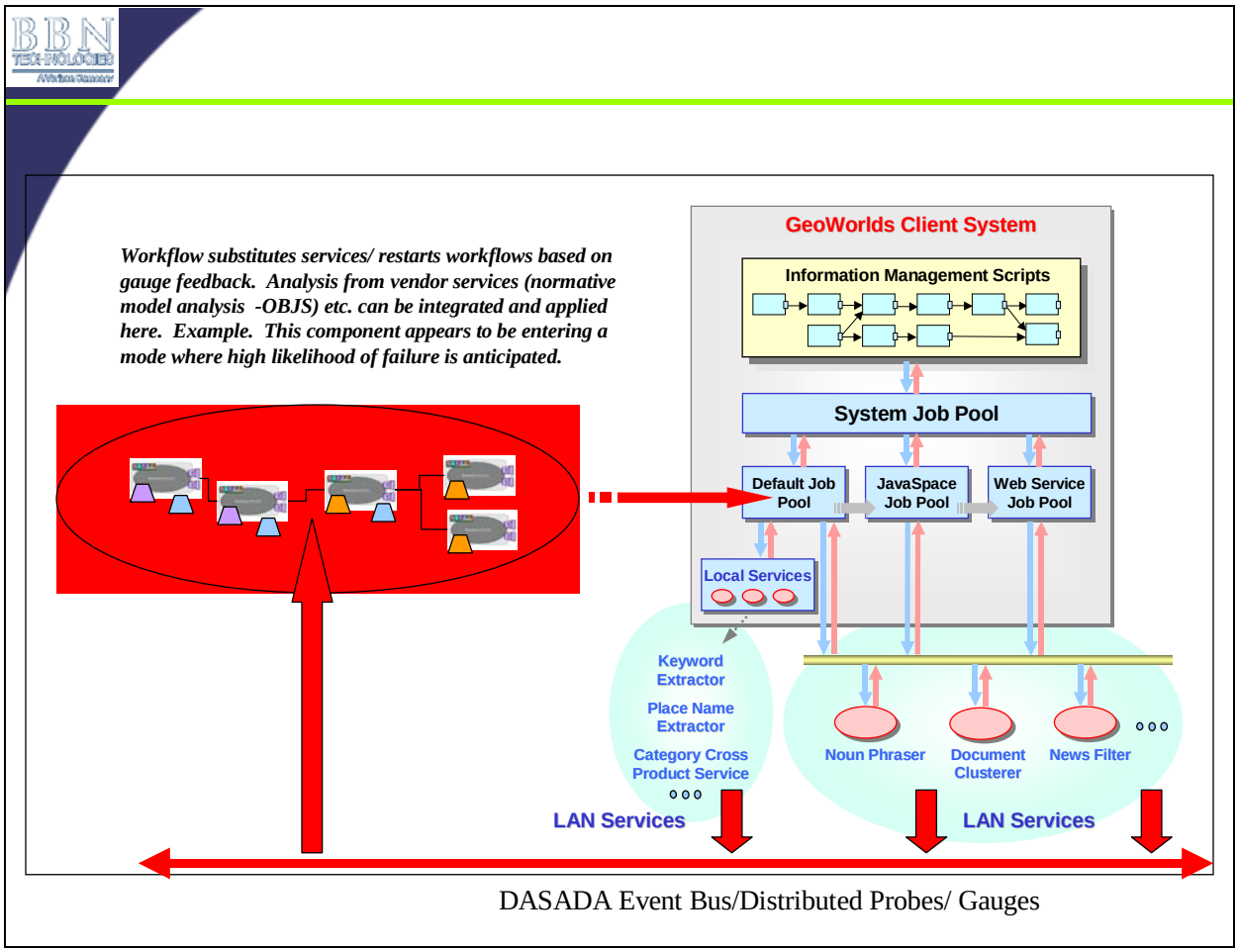

Figure 24. 2002 Demonstration system (idealized).

In 2002 we demonstrated how an Architecture Description Language (ADL) can be used to monitor and adapt service architectures from notification events. We demonstrated how capturing publish/subscribe events from the SC layers can then be converted "on-the-fly" into ACME ADL models. One of the important issues here was how to handle inconsistency in a stream of events as they poured into the Event Atlas. Given the distributed nature of an SC system, events would arrive out of order as well as some events may not appear at all (logging channels may be down or unreliable). There were two aspects of this challenge:

- Order the events

- Decide how long to wait for missing events
- Decide how to minimize architecture model impact when missing events

The 2002 version of the EventAtlas primarily focused upon the latter two concerns. It exposed a plugin model where we tested and planned to more fully explore use of Columbia University's EventDistiller to filter events (temporally) as a supporting service.

In the 2002 demonstration we used the ACME Architecture Description Language (ADL) [32] to represent an external model of the SC workflow. This model was built using publish/subscribe events captured from the runtime of the demonstration system - events were converted *on-the-fly* into ADL. By decoding the events in context of knowledge about the SC protocol – the EventAtlas (Figure 25) can generate and display ADL in ACME Studio [28] even when the underlying SC system is changing quickly.

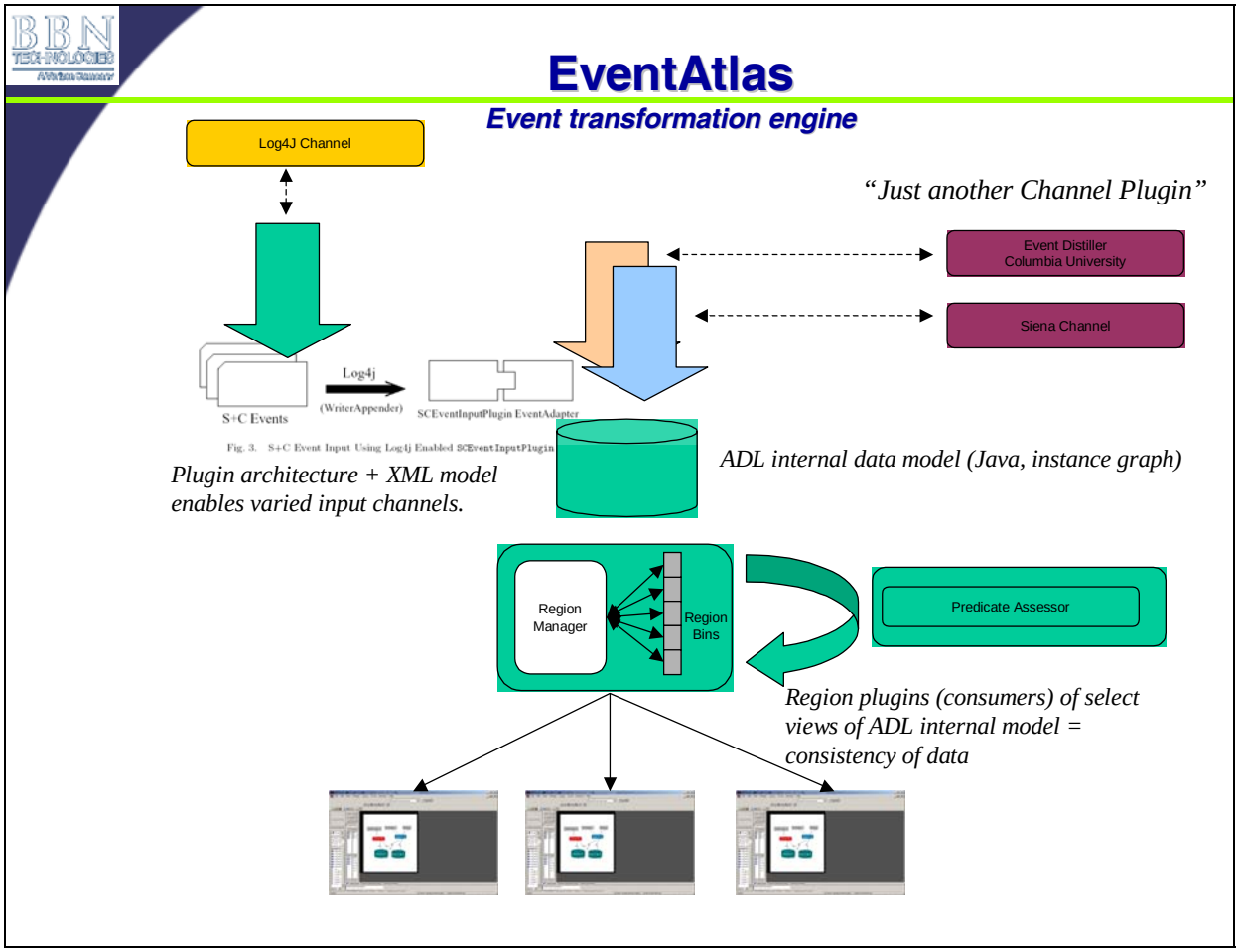

Figure 25 Service and Contract workflow events were captured using logging (Log4J) channels and coverted into ADL internal form. ADL internal forms were then translated on-the-fly into various external consumer formats (UI, ACME ADL).

In 2002 we further demonstrated how a distributed workflow memory system can be used to optimize performance (service discovery) and expectations using purely local mechanisms. Hints

were computed by the infrastructure from metrics that flowed back with the workflow Results. Hints were used to alter how future workflows would behave.

For the demonstration, *service invocation times* were flowed back along the workflow along with the Results – these were then observed by the SCConnector infrastructure elements at each agent. Hints would then be composed by the SCConnectors (default behavior); Hints would be attached to future workflows that passed by the agent. Like all Directives, Hints would flow downstream in the workflow from the point of insertion. In the 2002 demonstration, we also illustrated the concept of information decay – we illustrated that it could play in deconflicting Hints produced by many agents in a large system.

We further demonstrated the future role of the DASADA Probe Bus [3] - used in conjunction with "CDIGauges" to *laterally inhibit* substitute services within a single agent. The idea here was that services within an agent could compete in parallel and inhibit other poorer performing services as they progressed. Competing substitute services could effectively self-select a winner using a CDIGauge. CDIGauges would monitor a cluster of substitute services within an agent shut down the least effective substitutes and therefore conserving resources.

In 2001 we developed an infrastructure and strategy for dealing with service exceptions (e.g., failures, time-outs) through *service substitution*. In 2002 we continued with this strategy, but elaborated upon it, containing it within a more complete adaptive model. Specifically, in 2002 we layered a Directives mechanism that *influenced* the shape of the workflow during its construction and subsequent processing. (Note that a Directive can be either a Constraint or a Hint).

In 2001 we demonstrated how Constraints can stimulate the infrastructure to recruit other "Gauge" services and to verify some aspect of the operating environment [3]. In 2002 we added SC mechanisms to facilitate the automatic substitution of services that violate Constraints (e.g., timing expectations or feedback from a Gauge service); for example, when an *eligible* local substitute exists. The substitute is eligible in the sense that it has been previously Accepted (by the substitute Service Provider) and Contracted (by the SC infrastructure) for such a contingency.

In 2002 we generalized the Constraint mechanism so that Constraints became just one type of Directive. In 2002, a new kind of Directive, the Hint, was introduced. All Directives act as types of messengers – they are "morsels" of information that are propagated along the workflow as it self-assembles across the system. A single type of Constraint was developed and used to motivate an infrastructure evaluation of a Contract either just before (PRE) or after (POST) invocation of the contracted service (depending upon the Constraint parameters). This could as a side-effect lead to the recruitment of additional Gauge services, etc. [3]. Hints also shape the workflow during the service-assembly process.

For example, Hints can suggest service destinations. For example, based upon past experience, a Hint might suggest to an agents infrastructure about to issue a remote Request, that it should go to X instead of Y. Another type of Hint was used by the infrastructure to encode what it thought was a "reasonable" (from previous experience) invocation time associated with a particular service. For example, based on past performance, if service "Foo" takes about 400 millisecond to

invoke – then a Hint may be generated to indicate that a significant deviation from this might be grounds for timing-out a "Foo" service encountered.

Hints are tagged with a weight that reflects the strength of that Hint as suggested by its creator SC infrastructure at a specific agent. In the current system – Hints weights reflect the "quality" of the metrics used to compose that Hint.

Figure 26 depicts the 2001 demonstration scenario, where workflows "ebbed and flowed" trying to build structures in the "forward" direction (outwards from the root Request insertion point). As these workflows are wired-up, certain information, namely Directives (Constraints and Hints) travel with them. It is important to emphasize that from the point of insertion, Constraints and Hints travel downstream as the workflow is constructed. Directives can be inserted by either the infrastructure or by an external source actor (application or user).

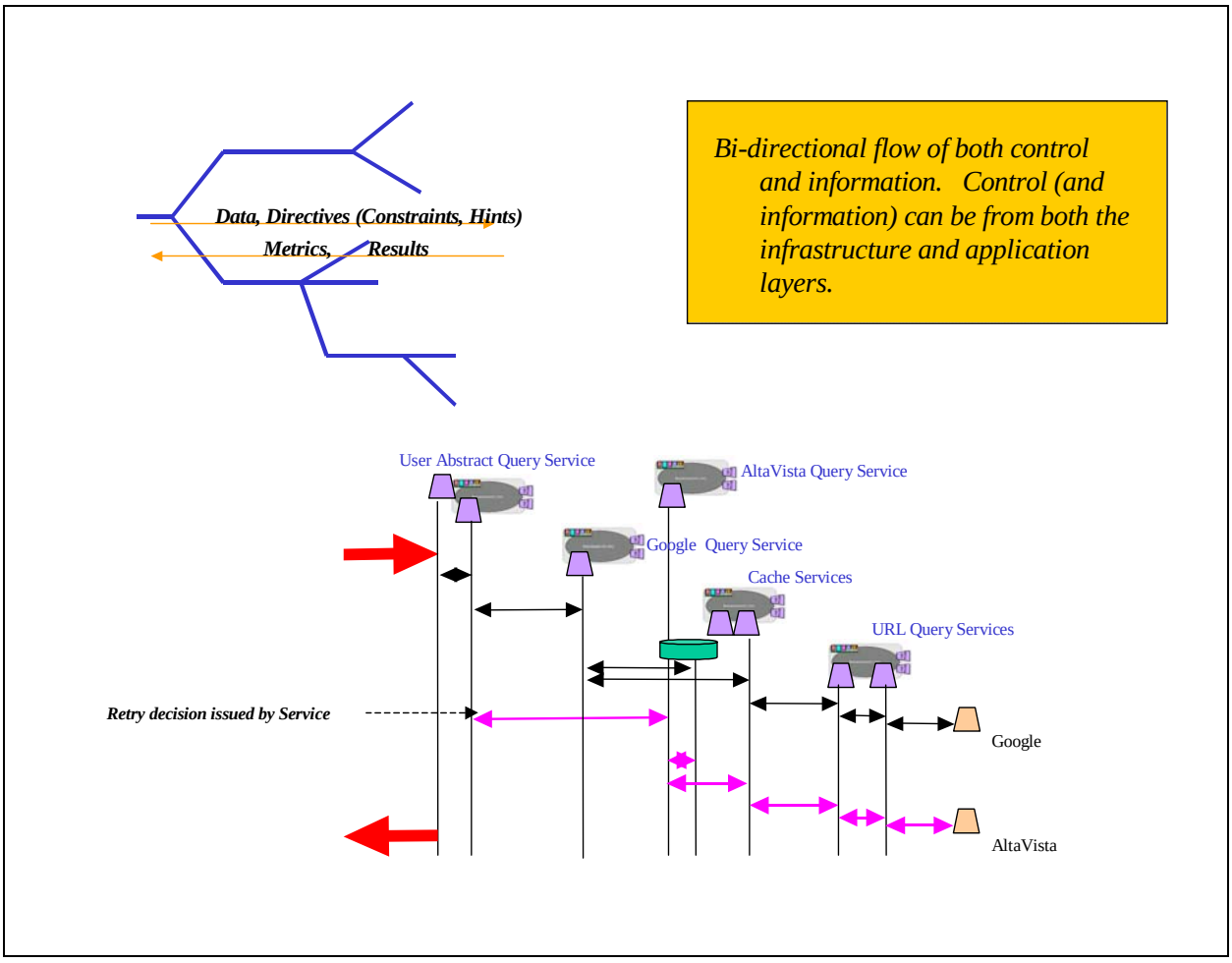

Figure 26: Service and Contract workflows create bi-directional information flows. Past performance shapes future performance.

BBN successfully demonstrated in 2002 an *Adaptive Mirroring* technique for use with SOS architectures. The key idea was to probe potential bottlenecks in a target system (GeoWorlds connectors) and route data to alternative services and channels when these GeoWorld connectors

failed. The alternative services and channels resided in the SC system. It was said to have mirrored the target services because it monitored the health of the system at specific points and was able to provide contingencies when needed. When the application failed at the probed points (e.g., via demonstration), then control and data was shifted to the alternative agent system. Connectivity into and out of the mirroring SC system was via Remote Method Invocation (RMI) and HTTP.

The 2002 annual demonstration targeted the GeoWorld's JobPool and external connectors to remote services at ISI in California for mirroring. The JobPool dispatched messages from the GeoWorld's client to supporting GeoWorld's services (local and remote).

In 2002 we probed the JobPool and monitored it with an SC system. When the JobPool was stressed (simulated), the SC system would switch processing to an alternate SC system. The alternate system had access to substitute instantiations of services used by GeoWorlds, such as ISI's Keyword Extractor and Noun Phraser. Results were returned from within the alternate system back to the original application.

Within the alternate SC system a large pool of substitute services using a "simulation wrapper". The wrapper allows us to configure each component instance with a different performance profile with different failure rates and invocation latencies. In this way, a small set of actual components (2) could be "cloned" into a larger number (>50).

The demonstration SC system was said to mirror the Geoworld's client in that for certain probed errors it would be able to find and swap-in use of services from a "pool" of substitutes. In the demonstration, services would be replaced because of exception noted at the JobPool or because of a Constraint violation (reflecting some bandwidth, load-balancing consideration, etc). In the 2002 demonstration, the pool of substitute services was predetermined. We have looked at enhancing future versions of our agent infrastructure to take advantage of those Cougaar [2] features that pertain to the dynamic loading and migration of components within a mirroring architecture.

Figure 27 is a schematic of the 2002 DASADA demonstration system. This system involved approximately 18 nodes and 150 components. There were three main scenarios exhibited:

- Geo Worlds (GW)-Keyword Extractor Scenario
- Simulation Scenario
- GW-Noun Phraser Scenario

The GW-Keyword Extractor Scenario featured provided alternative GW-Keyword Extractor services to GeoWorlds. During the demonstration, the GeoWorlds client was forced to fail using its own connectors to the remote (ISI) service. Within the SC system, two types of alternate GW-Keyword Extractor services were cloned to form two different "pools" of services. The different types of services were co-mingled in the different agents. This scenario showed how over time using Hints the SC system would learn to favor one pool of services over the other (the fast one),

however, as the load on the system would be throttled up, then when the supply of fast services would become saturated, then the slow ones would join the fray.

The GW-Noun Phraser Scenario emphasized how a CDIGauge could be used to successively prune a pool of contemporaneously executing services as they started to indicate failure (via a simple Gauge that measured logging warnings and errors). Failure was again induced by simulation. As described earlier, the CDIGauge was a stand-in for a future DASADA Common Gauge Infrastructure service.

The Simulation Scenario involved the most agents and used a number of simulated services to emphasize basic SC features for demonstration.

Significant of the SmartChannels composition was that the three threads co-existed on multiple hosts and Java Virtual Machines. Individual agents were largely unique (though not exclusively) to one thread or another. A top-level agent gated Requests by type to different thread groups. This is indicative of how a single physical implementation substrate can support many different functional threads.

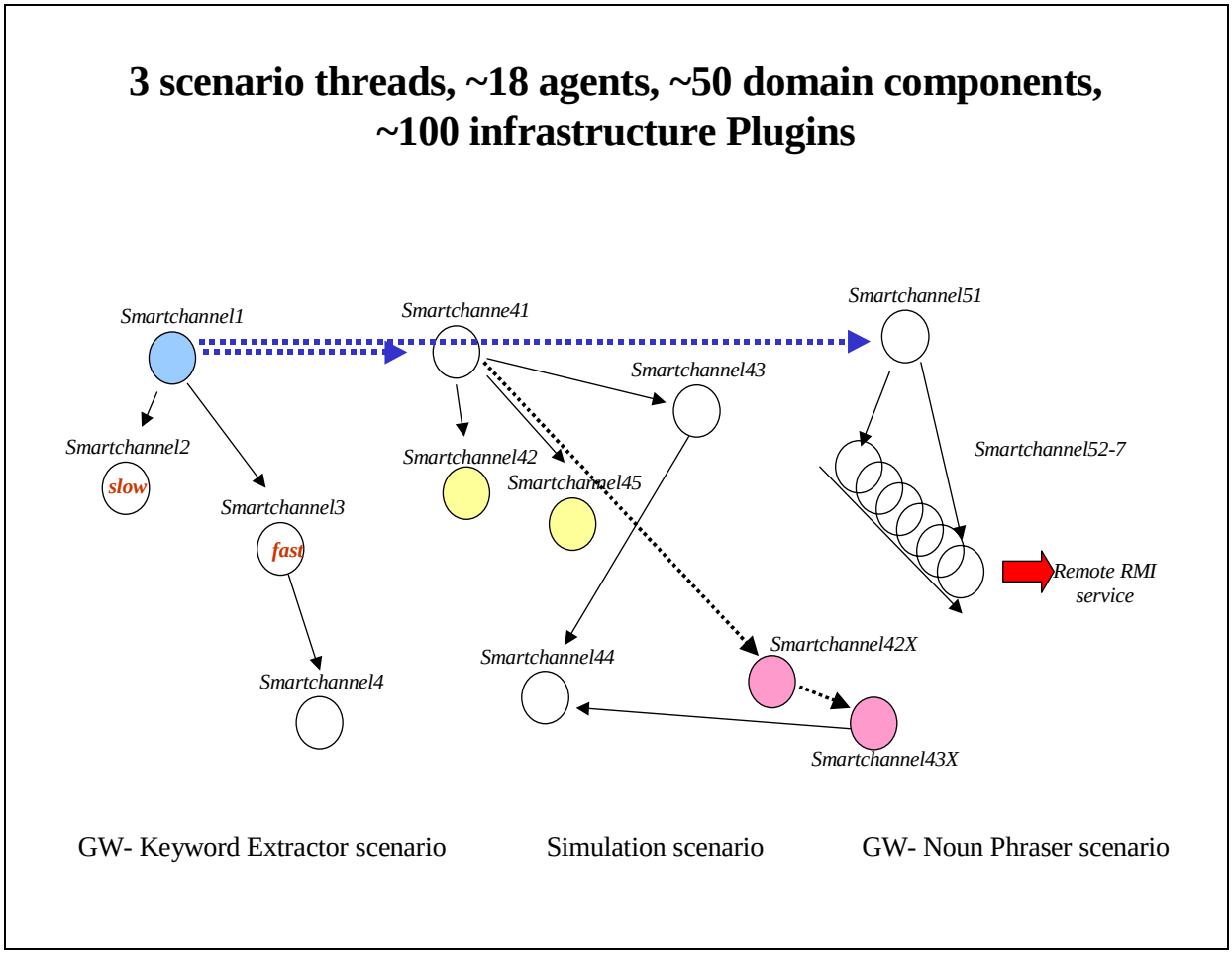

Figure 27 2002 DASADA demonstration system: 18 nodes, ~150 Plugins.

As described earlier, in 2001 we used a commercial hyperbolic visualization tool to view the interactions of the Service and Contract components during execution. While this viewer was useful for displaying and navigating a vast quantity of low-level SC events (publish subscribe events associated with SC LDM) it was less useful for a system of any size. A more abstract representation of architecture would be necessary. In 2002 we constructed an ADL model on the fly from these low-level publish/subscribe events. The events were fed to the EventAtlas tool and allowed us to build up an architecturally meaningful depiction of the interactions of services (Connectors and Components). Figure 28 illustrates this mapping (idealized).

In the 2002 demonstration, from the perspective of the architecture model, components and Service Providers were synonymous. Thus a graph of assembled Service Providers was represented by an ADL graph. However, if you recall Figure 27, just as a component, and a connector, can be an abstraction for finer software details – depending upon the desired fidelity of the model – so can components and connectors represent populations of services.

Under this contract we started to look at the idea of an adaptive multi-level ADL model for SC systems. Could not a component in an ADL representation model some grouping of committed Service Providers, depending upon what the purpose of that model (what it were attempting to communicate)? Furthermore, could these groupings be dynamically altered based on where the interest and attention of the consumer (viewer or application) was focused?

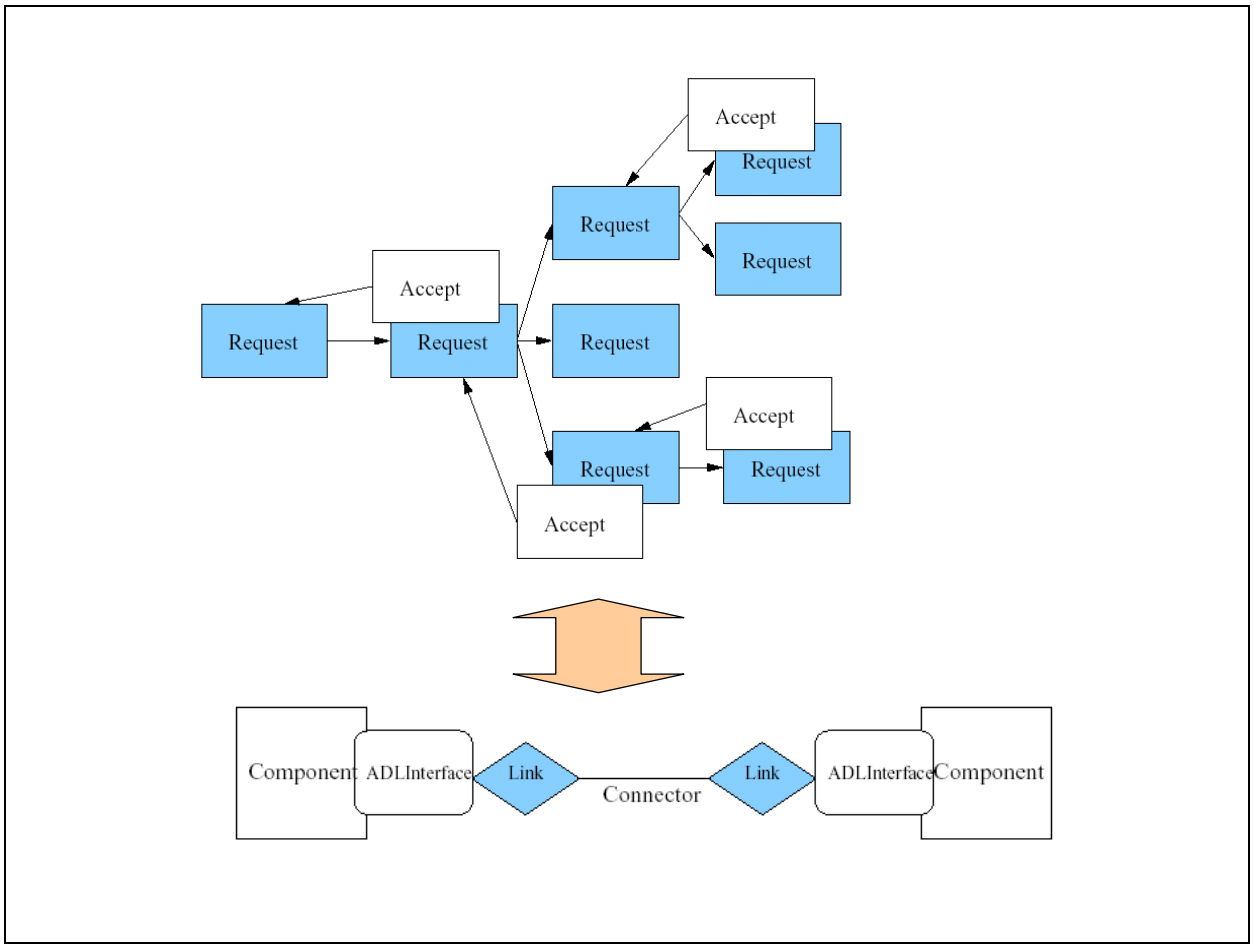

Figure 28 Architecture Description Language (ACME) used to provide more powerful action for visualizing they dynamics of an SC architecture.

5. Future Work

Under this contract we implemented the AAI Toolkit – a collection of research capabilities for building adaptive SC architectures. Future work might continue along two basic thrusts:

1. Other agent frameworks.

2. Exciting research directions.

We would like to experiment with other agent frameworks to insure the mobility (transparency) of the SC protocol across implementations. Furthermore, we would be interested in a "lighter implemention" of a framework for "what if" experimentation – a framework that can be coded and manipulated at a higher level than with Java (Cougaar). While such a light framework would likely not be of sufficient quality to transition to operational environments, it would hasten experimentation and testing of new ideas. Then once ideas are firmed up they can be migrated in the AAI Toolkit (Cougaar) baseline.

A number of exciting research directions were described earlier in the paper. These are indicative of the potential (and research difficulty) with identifying and describing technologies useful to adaptive system design. A few are highlighted below:

- *Enhancement of the SC Protocol to allow global stabilization of plans before execution.* The research challenge is to do this *via negotiation* of agents and services, in the spirit of "consensus building" vs. using a centralized device (lock or control).

- *Development of Adaptive Service Neighborhoods.* What distance may a service reach out? What is the measure of distance? Is it the workflow graph distance, or some other measure of the separation within a process?

- *Synergy via interaction with 3rd party Quality of Service and monitoring systems.* This would build upon the idea that Gauges can in fact act as a proxy for an external Gauge service (such as DASADA Runtime Gauge Infrastructure).

- *Extend the SC protocol to enable multiple parallel Executor Plugins within a single agent.* What is the proper granularity of *agent vs. service*, and how can we *quantify* this relationship? Should agents encapsulate many services or should there be many agents? Ultimately we feel the answer depends upon the application and the granularity of the service/components.

- *Foviating views of SC architecture*[1]. We started to look at the idea of an adaptive multi-level ADL model for SC systems. Could not a component in an ADL representation model some grouping of committed Service Providers, depending upon what the purpose of that model (what it were attempting to communicate)? Furthermore, could these groupings dynamically change based on where the attention of the consumer (viewer or application) was focused?

[1] Data from head position trackers is used to compute the stereoscopic display images in immersive Virtual Environments. Gaze tracking can be used to further improve the design of visual displays. The resolution of the displayed image can be decreased as pixels get further away from gaze direction since the resolution of the retina decreases towards the periphery of field of view (the so-called foviating effect).

6. Supporting Investigations

A number of supporting/lesser investigations were performed during the period of performance of this program. The purpose of these investigations was to illustrate the versatility of the Service and Contract protocol and the AAI Toolkit for use within the DASADA program:

- The Service and Contract protocol is *event based* – this enables the system to react to changes in its structure and/or environment more effectively.
- A mature and well-defined component model (leveraged from the underlying Cougaar infrastructure).
- The Service and Contract idiom based on Blackboard objects allows easy extension of the language.

6.1 Capability-Secure Data Access for Trusted Service Coordination in Cougaar Societies

Described below was a prototype SC system that tested a capability-secure technology [1] to securely manage fine-grained access of data and services [2]. The introduction of a capable-secure protocol into the SC system was hypothesized to better enable fine-grained exchange of trust within the system.

6.1.1 Service and Contract Distributed Workflow Introduced

As noted previously, the Service and Contract (SC) workflow technology was used as means to reliably assembly of services within a distributed publish/subscribe framework.

This investigation looked at whether use of a fine-grained capability security model was compatible with an SC protocol. This would contrast with a Access Control List (Matrix) security model. The following advantages were hypothesized:

- Fine-grained privilege model
- Privileges that can be tied to specific pieces of data (objects on the Blackboard)
- Transferable and revocable privileges

We were exploring the idea that an ACL model, in itself, might not be fine-grained enough for a very large SC system operating in a dynamic trust environment. Thus, it may be too costly to use ACL's to grant partial and temporary access to *some* data on an agent Blackboard (e.g., defining an ACL to match every use/access-pattern, etc.). We were looking for an approach to manage privileges that better scales to a dynamic SC environment where revocation and alteration of privileges can be entertained without continuous ACL modifications.

6.1.2 Capabilities Security Model

We explored what it would mean to use a capabilities model (based on E [17]) for mediating object-level access within an SC system. Objects on the Blackboard that were granted external

access were assigned a cryptographic Uniform Resource Identifier (URI) handle. Handles are distributed as cryptographic URIs that make the names hard to guess what they reference. This enabled the following:

- Agents may share with other agents cryptographic URI references to objects they encapsulate. These URIs may reference local data or may authorize the reservation of a local service by another agent.

- URIs may be de-referenced over a secure channel to a Cougaar integrated E [22] server. SC Cougaar agents use a specialized PlugIn (SCConnector) to encapsulate an E server and manage interactions with the SC infrastructure.

- URIs can be invalidated at the agent who owns the referenced data or service. In so doing an agent wishing to revoke another agent's visibility or access to that object can do so. Because an arbitrary number of cryptographic URIs can be issued to reference any piece of data, individual references can be selectively revoked without repealing everyone's access.

6.1.3 Capability-secure Data Access and Cougaar

The addition of capable-secure features to the SC infrastructure was hypothesized to be useful in enhancing agent collaborations as follows:

- The fine-grained transfer of *trust* permits infrastructure strategies where visibility and access to distributed data and services can be limited to a "need to know basis" (see Figure 29).

- Delegation of trust to third parties is permitted (see Figure 30).

- The fine-grained revocation of trust (capabilities) is necessary for dynamic systems (see Figure 31).

Within the context of the SC infrastructure, a minimal prototype of a new variant of the *SCConnector* Plugin (SC infrastructure) was developed and evaluated. This SCConnector integrated a capability-secure server that could on-demand bind Cougaar Blackboard objects to "capable-secure" collections of objects to other agents requesting visibility into the data over HTTP. Note that this new functionality was orthogonal to the SC purpose of the SCConnector. It was a convenient new piece of test functionality that was assumed by the SCConnector.

It was shown that we were able to assign "capable" URIs to agent Blackboard objects. A capable URI is a cryptographic reference to a capable-secure object encapsulated by that agent. It was shown that this new SCConnector was able to act as a server to answer calls for access to URI referenced data. In this way service reservations and data can be exchanged.

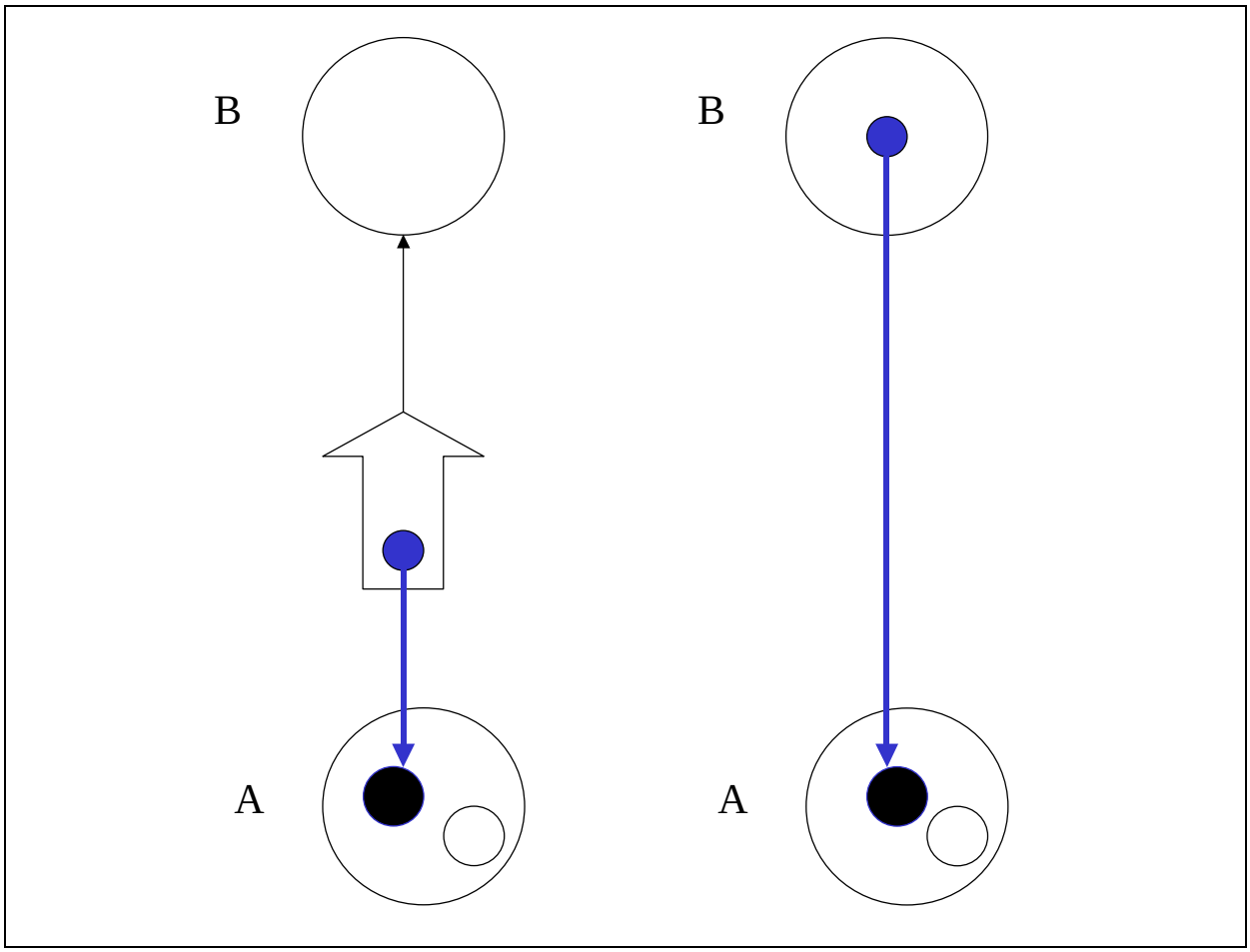

Figure 29 Basic Case. Capability is transfered (blue dot) from agent A to agent B for visibility into select objects (black circle).

In the figure above, note that visibility into agent Blackboards is scoped by agent boundaries. Agents share Capability URIs. An agent that holds a Capability URI is able to access the data object to which the object refers. In the E language [20], a Capability URI may look like:

```
cap://128.33.238.61:1106/PiBnoCERpHxbZEzjvPVtd8FO19G/Ujh2fdgarH1unAdtuRU483XFb
2S
```

We had prototyped an AAI system to use Cougaar PlugIn encapsulation of an E server to manage distributed external access into Cougaar runtime objects using these encrypted URIs.

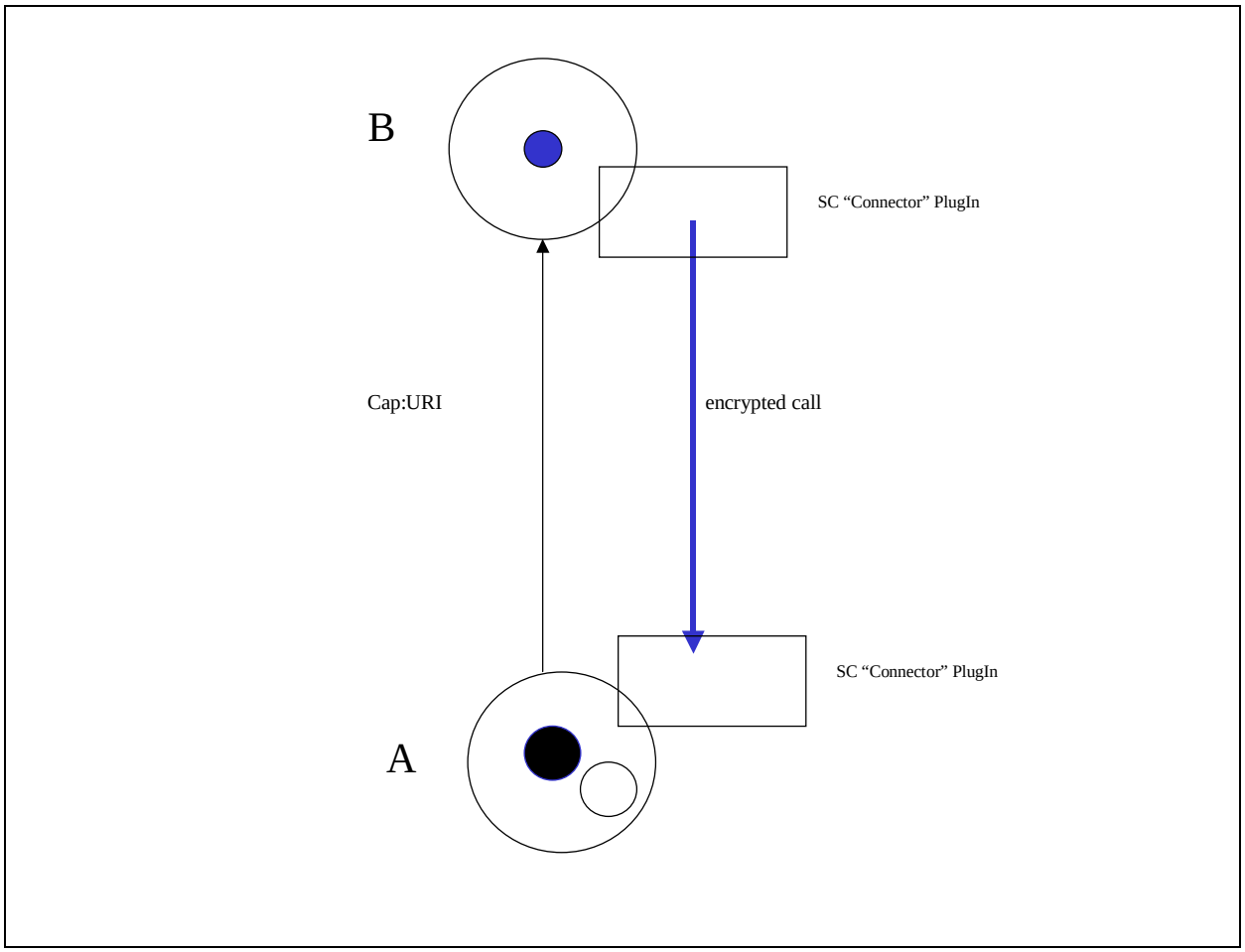

Figure 30 Elaboration of mechanism: Service and Contract infrastructure.

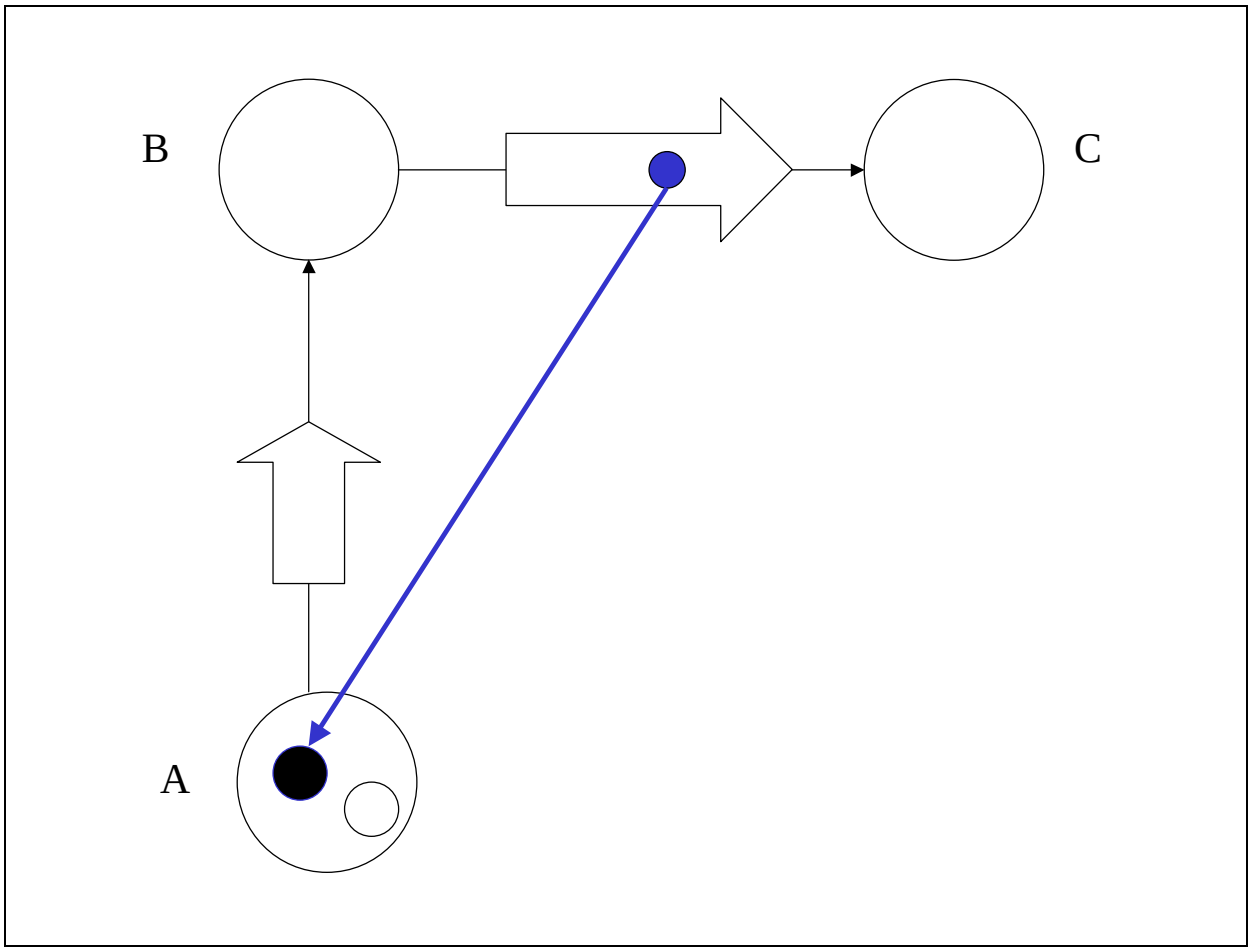

Figure 31 Agent B delegates capability to Agent C.

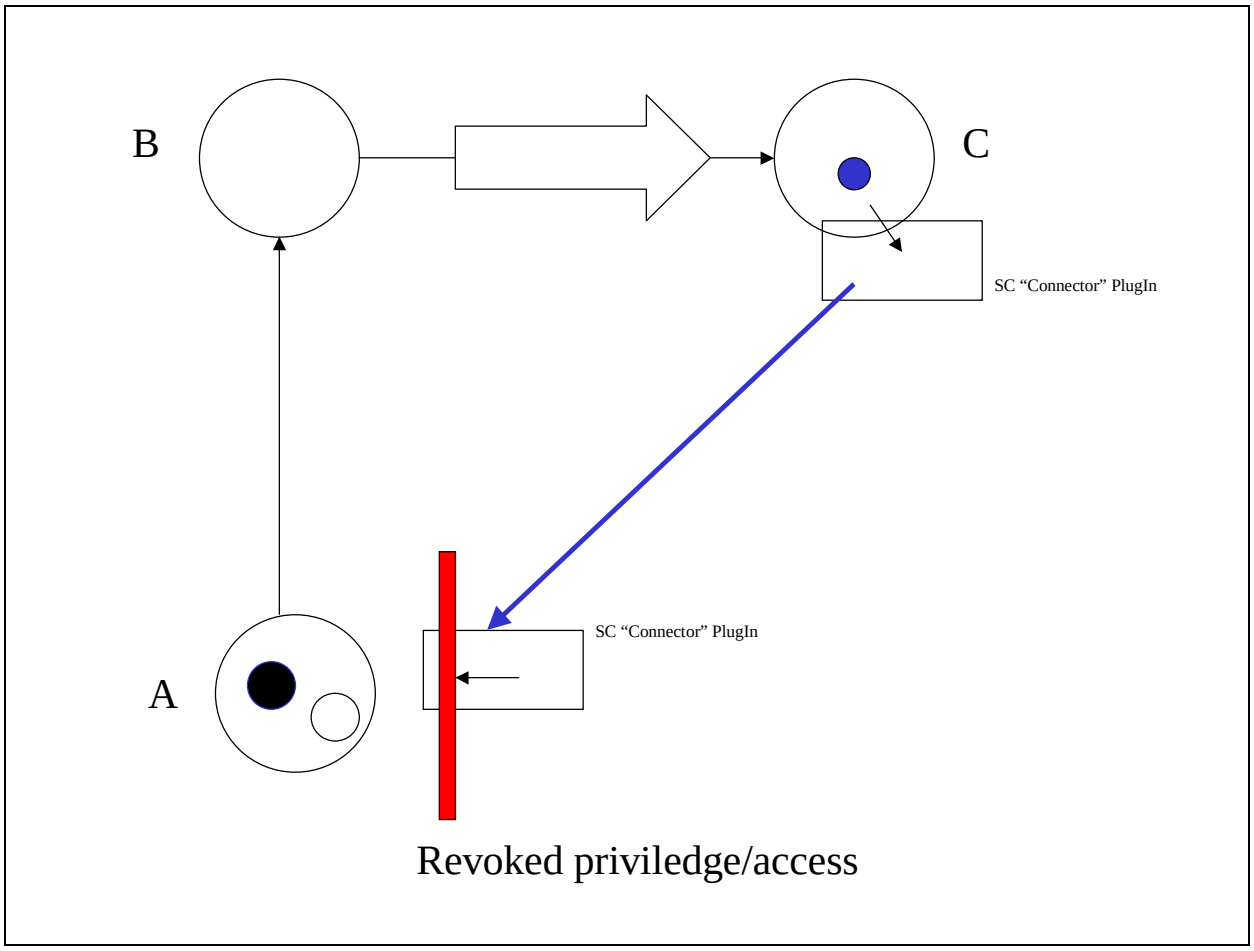

Figure 32 Revoking a priviledge (URI blocked)

```
String [] args = new String[2];
        args[0] = "-De.home=\"..\\..\\..\\e"; //"-De.home=\"c:\\erights.org\"";
        try{
            PrintWriter stream = new PrintWriter(System.out);
            LineReader reader = new LineReader();
            reader.setSource(eSource);
            System.out.println("[ESPPlugIn.execute()]  Starting Interp.");
            TextWriter tw = new TextWriter(stream);
            myRunner = new Runner("My Main Vat Thread");

            synchronized( myLock )
            {
               myInterp = Interp.make(args,
                        null,
                        tw,
                        tw,
                        myRunner,
                        reader,
                        null,
                        false);
             }
             stream.flush();
             if (false == myInterp.interpret(! myInterp.getInteractive())) {
                    new RuntimeException("Failed to interpret!");
             }
             stream.flush();

         } catch (IOException ex ) {
            ex.printStackTrace();
        }
```

Figure 33 Sample code. E interpreter (version 8.9.1b) integration point with Cougaar PlugIn. The SCConnector PlugIn is an "E" server executing an E server script. Objects from the agent runtime (via PlugIn) are bound to server variables and capability URIs are returned.

6.2 Service Advertisement using Multicasting

A simple experiment based on the Reliable Multicast Framework and Reliable Adaptive Multicast Protocol (RMF/RAMP) product distributed by TASC/Northrop Grumman [26] was conducted. The purpose of this experiment was to test/show how the Service and Contract infrastructure can be specialized with new capability (multicast neighborhoods) and conduct themselves in an SC system. An emphasis of our approach was to encourage embellishment of a foundational SC protocol with new behaviors. The claim was that diversity should not necessarily undermine the core capability of the system.

Ostensibly, the experimental hypothesis was whether a multicast "service advertisement" is efficient means of finding and recruiting services in the local neighborhood of an agent.

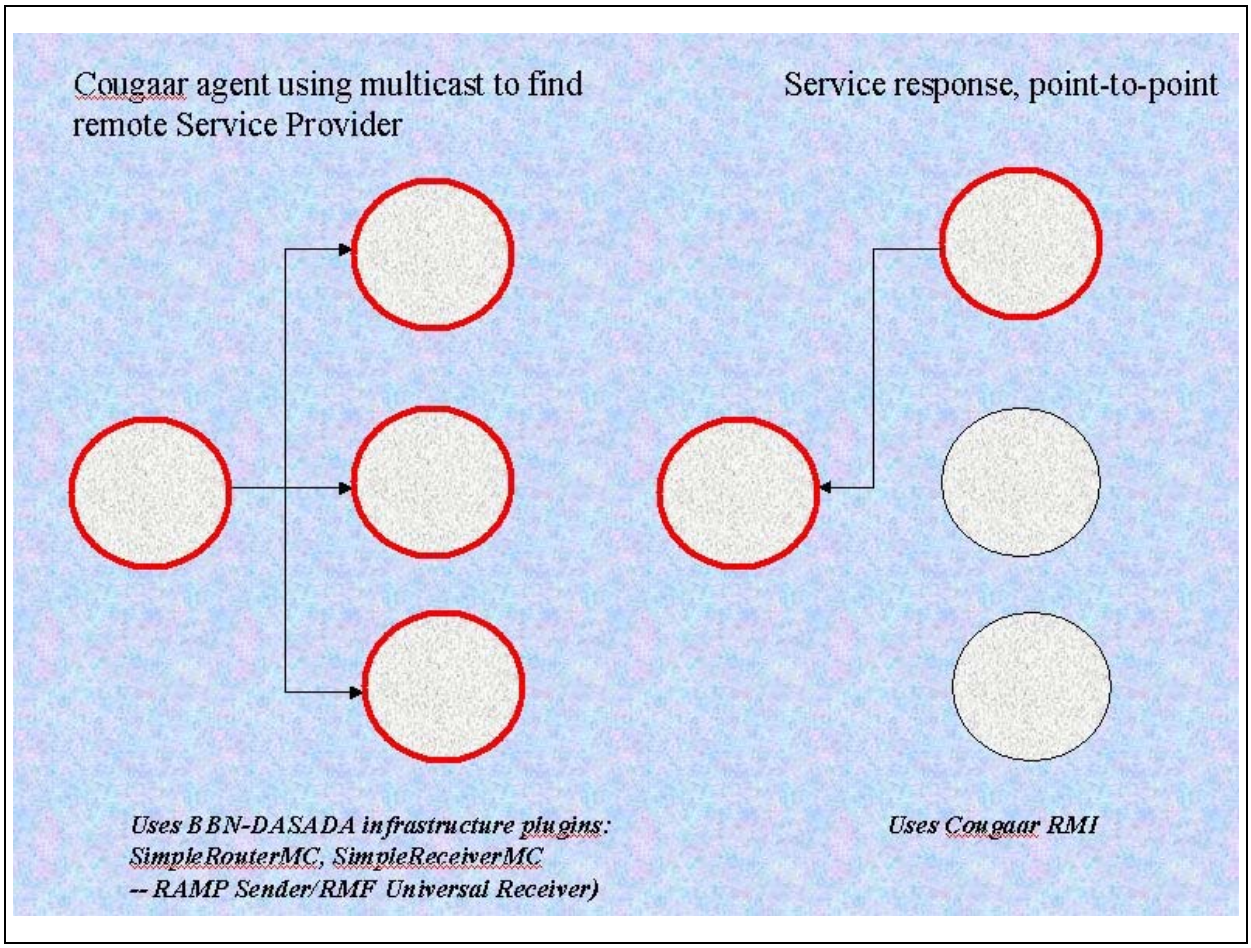

Figure 34. Early experimentation model emphasizes multicast for "service advertisement" vs. content distribution.

Figure 34 above illustrates the optional use of reliable multicast (via Router infrastructure PlugIn) during the fan-out "service request" phase of constructing distributed workflows. An agent that cannot recruit a local service to satisfy a request for service will broadcast the request to other agents in case those other agents are able to do so. Should one or more agents be able to perform the requested service from within their local context, they will respond to the sender (on a point-to-point basis).

The SC Router is part of the "Service and Contract" infrastructure PlugIn. Earlier we described how this is the infrastructure component responsible for deciding and executing upon a number of S+C infrastructure functions. It decides, specifically, whether, when, and to who to send service requests that aren't satisfied locally. In our experiment, we specialized some of the SCRouter used by the agents so that they multicast in the forward direction (to solicit new services). The default SCRouter used point-to-point messaging amongst potentially many agents. The return channel was point-to-point amongst a pair of agents (more precisely the return channel is handled by an SCConnector Plugin – a component that operates in tandem with SCRouter). In the forward direction (workflow downstream) solicitations for services are sent. In the reverse direction (workflow upstream), results and metrics are returned.

Agents use SCRouterMC and SCConnectorMC Plugins to communicate to others in the multicast group. This is instead of the SCRouter and SCConnector Plugin pairs. However, what is powerful is that a single agent can have both sets of Multicast plugins participating in the same multicast group "listen" to the same group Class D IP address.

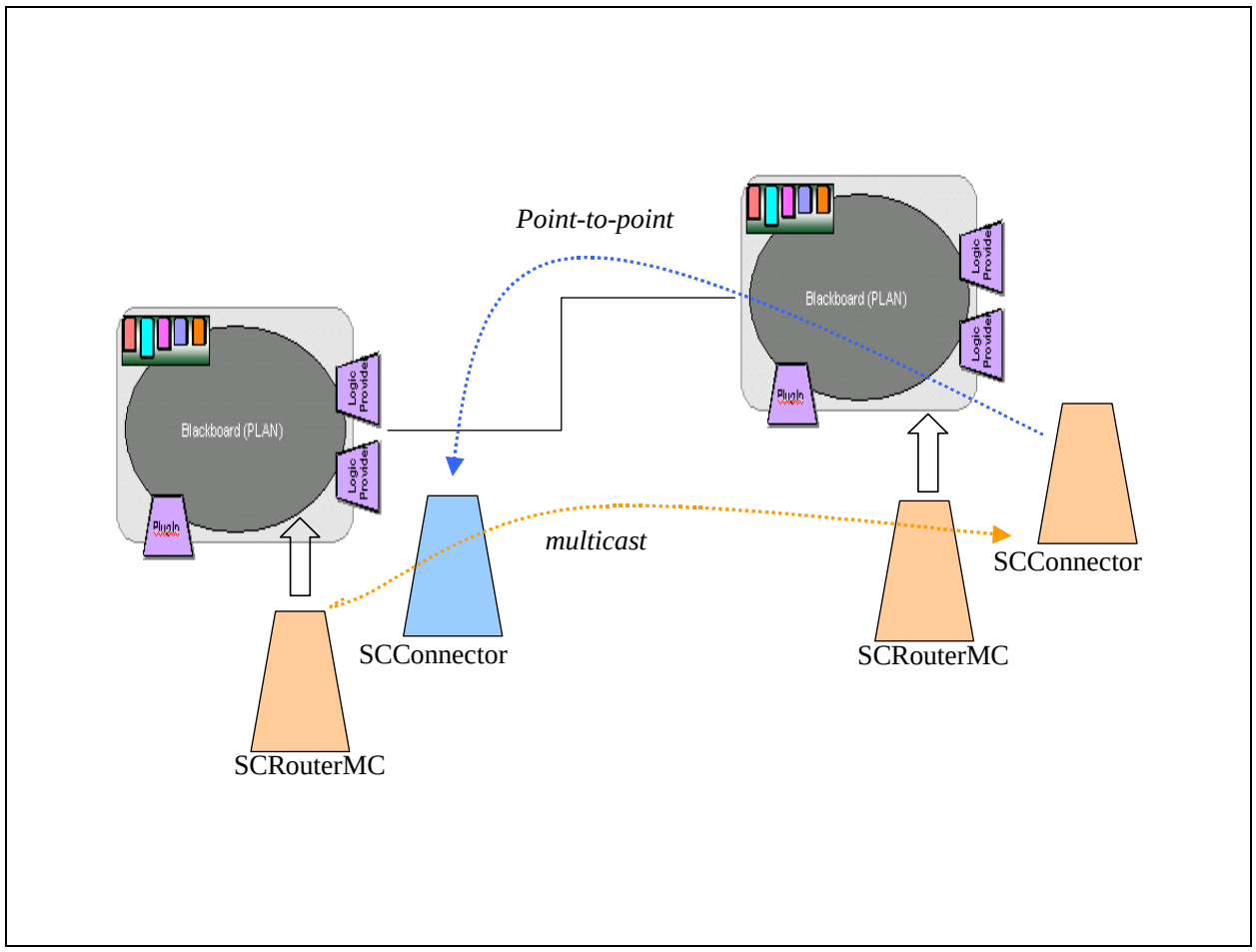

Figure 35 Optional multicast inserted as PlugIns. Multicast "routers" are specialized SimpleRouter infrastructure PlugIns.

This experiment show how different infrastructure capabilities interrelated by module relationships can be used to construct diverse society of agents spanning very different connectivity types and profiles (Figure 36). Factoring infrastructure behavior into components enables construction of diverse functional communities still able to interoperate. In this experiment, the simple SimpleRouter infrastructure Plugin was extended via the SimpleReceiverMC. In this way, it was possible to have two communities: One in a multicast group, one not. Furthermore, because of the ability of infrastructure plugins to co-exist within a single agent – an agent can be communicating with its neighborhood via a diverse range of connections.

This particular idea was used in the 2002 DASADA demonstration – in that case though, the specialized Plugins communicated with the DASADA Gauge Bus (described earlier).

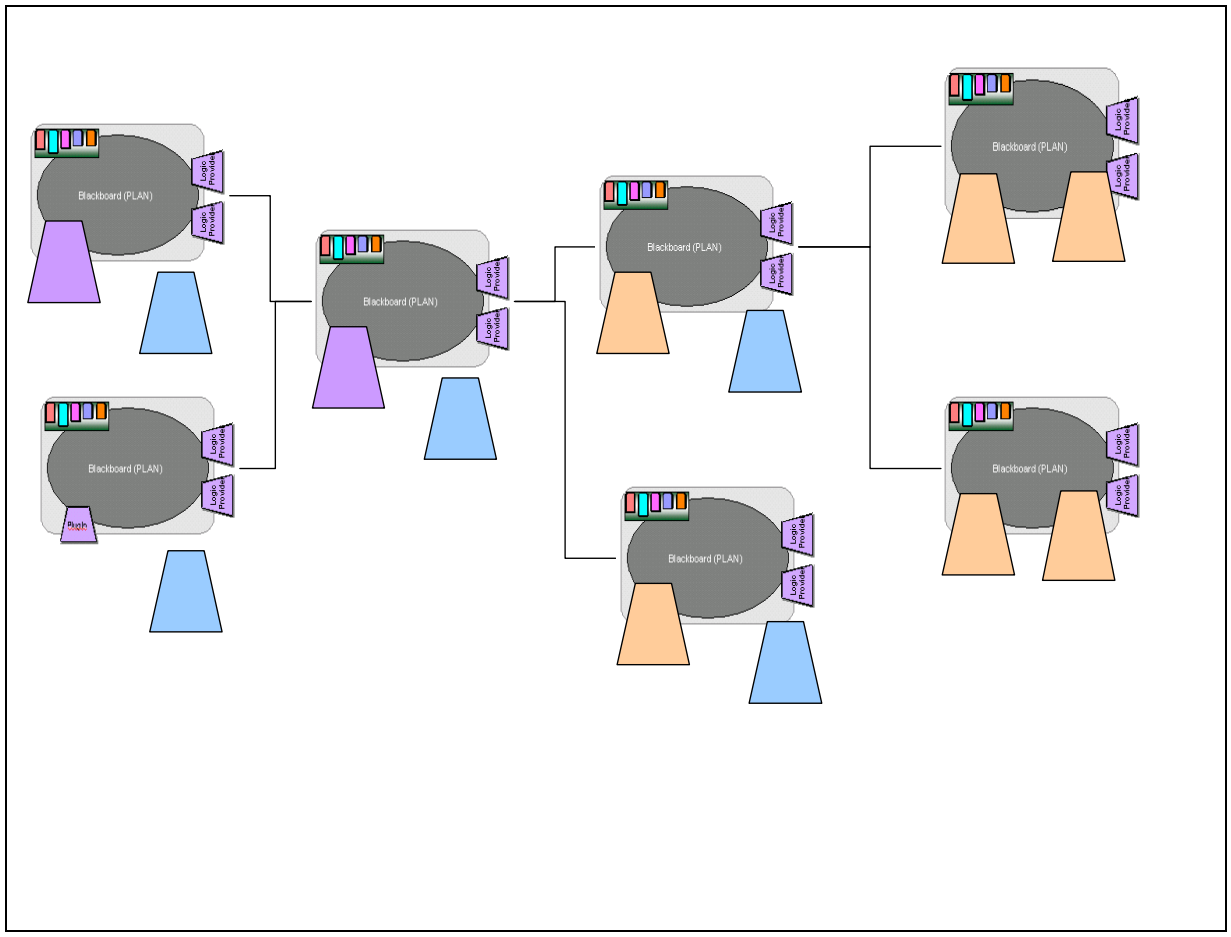

Figure 36 Modularizing infrastructure into Plugins permits diverse societies to be constructed. For example, working groups within a society can choose different communication mechanisms to suit their context.

References

[1] Distributed Operator Model Architecture, on the website of http://omar.bbn.com

[2] BBN Quality of Objects Website. http://www.dist-systems.bbn.com/tech/QuO

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

[4] B. Schmerl and D. Garlan, Exploiting architectural design knowledge to support self-repairing systems, in *14th International Conference on Software Engineering and Knowledge Engineering*, July 2002. http://www-2.cs.cmu.edu/~able/publications/seke02/.

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

[6] M. Greaves, Ultra*Log. http://www.darpa.mil/tto/programs/ultralog.html.

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

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

[9] Service and Contract Developer's Guide. http://aai.bbn.com/cougaar/SCLanguageDocumentation.pdf

[10] 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.

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

[12] DARPA, *Control of Agent Based Systems*. http://coabs.globalinfotek.com/.

[13] *Openwings.org*. http://www.openwings.org/index.html.

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

[15] Resource Description Framework. http://www.w3.org/RDF/.

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

[17] Capability security introduced, on the http://www.erights.org/elib/capability/index.html website.

[18] Cognitive Agent Architecture Open Source Website, on the http://www.cougaar.org website.

[19] BBN-DASADA, on the http://aai.bbn.com/ website.

[20] Reliable Recruitment and Assembly of Peer-to-peer Services and Distributed Workflow, on http://aai.bbn.com/cougaar/CDSA_final.pdf website.

[21] Open Source Distributed Capabilities, on the http://www.erights.org/ website.

[22] The E Language in a Walnut, on the http://www.skyhunter.com/marcs/ewalnut.html website.

[23] Henry Levy, Capability-Based Computer Systems, Digitial Press, Bedford, MA (1984).

[24] E-speak Architectural Specification, on the http://www.e-speak.net/library/pdfs/a.0/Architecture.pdf website.

[25] Alan H. Karp, Rajiv Gupta, Guillermo Rozas, Arindam Banerji. Split Capabilities for Access Control, on the http://www.hpl.hp.com/techreports/2001/HPL-2001-164.html website.

[26] Reliable Multicast Framework and Reliable Adaptive Multicast Protocol (RMF/RAMP), on the website of http://www.tascnets.com/newtascnets/Software/RMF/

[27] Java Remote Method Invocation. On the website of: http://java.sun.com/products/jdk/rmi/

[28] The *Acme* Architectural Description Language. On the website of: http://www-2.cs.cmu.edu/~acme/

[29] The Brain Inc. http://www.thebrain.com/Default.htm

[30] The J-Scheme Web Programming Project. http://jscheme.sourceforge.net/jscheme/mainwebpage.html

[31] Apache Software Foundation, The Jakarta Project, http://jakarta.apache.org/

[32] *Acme* Architectural Description Language. http://www-2.cs.cmu.edu/~acme/

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