To appear in the procdeedings of 2004 IEEE International Conference on Web Services, July, 2004, San Diego

Team-based Agents for Proactive Failure Handling in Dynamic Composition of Web Services

Xiaocong Fan, Karthikeyan Umapathy, John Yen, and Sandeep Purao
School of Information Sciences and Technology
Penn State University
University Park, PA 16802

ABSTRACT

Currently web services composition problems are addressed using AI planning techniques. The team-based approach, with emphases on the sharing of mental models and proactive collaboration, provides an alternative to current static approaches to web service composition. The approach provides clear advantages for proactive handling of failures that may be encountered during execution of a complex web service. The paper proposes a generic framework for dynamic web-service composition, and extends the CAST architecture to realize the framework.

Keywords
Dynamic Web Service Composition, Failure Handling, Intelligent Agents, Proactive Monitoring, CAST Agents

INTRODUCTION

The web services composition problem has been recognized to include both the coordination of sequence of services execution and also managing the execution of services as a unit [Pires 2002]. Monitoring of the execution and exception handling for the web services must, therefore, be part of an effective strategy for web service composition [Oberleitner 2003]. However, much current work in web service composition continues to focus on services discovery and the services planning stage. We argue that work from research in team-based agents can be leveraged to bridge this gap between planning and execution of web service composition.

The objective of this paper is to investigate how a team-based agent architecture—CAST, can be used for integrating planning, execution and monitoring of composite web services with a view to proactively dealing with failure handling. A successful approach in this direction can provide new ideas for responding proactively to changes in the environment or capabilities of web services at runtime so that the execution can be better monitored to achieve better QoS (quality of service).

CAST (Collaborative Agents for Simulating Teamwork) [Yen 2001] offers a suitable alternative for dynamic web services composition because of two properties they exhibit. First, CAST agents are designed to work collaboratively in a changing environment using a shared mental model of the environment. Second, CAST agents are designed to proactively inform each other of changes in the environment that they perceive to handle any exceptions that arise in achieving a team goal. By collaboratively monitoring the progress of a shared process, a team of CAST agents can not only initiate helping behaviors proactively but also can adjust their own behaviors appropriately to the dynamically changing environment.

In this paper, building on the CAST model [Yen 2001], we first propose a generic team-based agent framework for dynamic web-service composition; and then extend the existing CAST architecture to realize the framework.

COMPOSING WITH TEAM-BASED AGENTS

The framework we propose distributes the web service composition task – planning and execution – to a team of agents. We first describe the capabilities of team-based agents, followed by how they form and collaborate in teams to achieve dynamic web service composition.

Team-based agents

A team-based agent A is defined in terms of (a) a set of capabilities (service names), denoted as C_A, (b) a list of service providers SP under its management, and (c) an acquaintance model M_A (a set of agents known to A, and their respective capabilities: $M_A=\{<i,C_i>\}$).

The agents, thus, play multiple roles. First, as a *service manager agent*, an agent A *knows which* providers in SP can offer a service S ($S \hat{I} C_A$), or at least *knows how* to find a provider for S (e.g. by searching the UDDI registry) if none of the providers in SP are capable of performing the service. Services in C_A are *primitive* to agent A in the sense that it can directly delegate the services to appropriate service providers. Second, as a *composer agent*, an agent is expected to compose a process using the known services to honor a user's request that falls beyond its capabilities, that is, for "*abstract* services."

The set of acquaintances, M_A, thus, forms a community of contacts available to an agent. This additional, local knowledge supplements the global knowledge about publicly advertised web services (say, on the UDDI registry). The acquaintance model is dynamically modified based on the agent's collaboration with other agents (e.g., assigning credit to those with successful collaborations).

Responding to request for a complex service

An agent, upon agreeing to honor a complex service request, initiates a team formation process:

(1) Upon receiving a request for service *S* (with, say, constraints on service quality), an agent (say, *C*) adopts "offering service *S*" as its persistent goal (see figure 1).
(2) If $S \hat{I} C_C$ (i.e., *S* is within its capabilities), agent *C* simply delegates *S* to a competent provider (or first finds a service provider, if no provider known to *C* is competent).
(3) If $S \ddot{I} C_C$ (i.e., agent *C* cannot directly serve *S*), then *C* tries to compose a process (say, *P*) using its expertise and the services in $C_C \cup \bigcup_{<i,C_i> \in M_C} C_i$ (i.e., it considers its own capabilities and the capabilities of those agents in its acquaintance model), then starts to form a team:

(i) Agent *C* identifies teammates by examining agents in its acquaintance model who have the capability to contribute to the process, i.e. $A \hat{I} M_C$, and $S_P \cap C_A \neq \phi$, where S_P is the set of services used in process *P*.
(ii) Agent *C* chooses willing and competent agents from M_C (e.g., using contract-net protocol [Smith 1980]) as teammates, and shares the process *P* with them with a view to working together as a team jointly working on *P*.

(4) If the previous step fails, then agent *C* either fails in honoring the external request (is penalized), or, if possible, may proactively *discover* a different agent (either using M_A or a using UDDI) and delegate *S* to it.

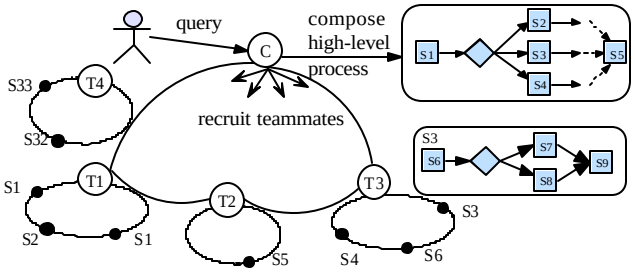

Figure 1: Dynamic Team Formation

Executing a complex service

Following the formation of the team the collaborating agents can, then, play a proactive role in executing a complex service. The collaborative monitoring of the shared high-level process will result in several kinds of proactive collaboration from each agent (see Figure 2).

Proactive Service Discovery. Let's consider a team agent *T2* is responsible for contributing service *S5*. If the service *S5* ceases to be available before the scheduled execution, *T2* will proactively attempt to discover a new provider for service *S5*.

Proactive Service Delegation. Suppose agent *C* chooses *S3* as the successive of *S1*, and *S3* itself is a complex service for *T3*, who composes a process for *S3* as shown in Fig. 2. Even though *T3* can perform *S6*, *S7-S9* are beyond its capability; it has to form another team and delegate the services to the recruited agents (i.e., *T6*). It might be argued that agent *C* would have generated a high-level process with more detailed decomposition, say, the sub-process generated by *T3* were embedded (in the place of *S3*) as a part of the high-level process. If so, agent *T6* would have been recruited as C's teammate, and no delegation would be needed. However, the ability to derive a process at all decomposition levels is too stringent a requirement to place on any single agent because it will lack knowledge that may be available to its teammates.

Proactive Information Delivery. Proactive information delivery occurs in the following situations. (i) There are critical choice points where several branches are specified, but which one will be selected depends on the known state of the external environment. Thus, teammates will proactively inform the team leader about those changes in states that are relevant to its decision-making. (ii) Upon making a decision, other teammates will be informed of the decision for them to better anticipate collaboration needs. (iii) A web service may fail due to many reasons. The responsible agent should proactively report the failure of services to the leader so that the leader can decide how to respond to the failure: choose an alternative branch or request the responsible agent to re-attempt the service from another provider.

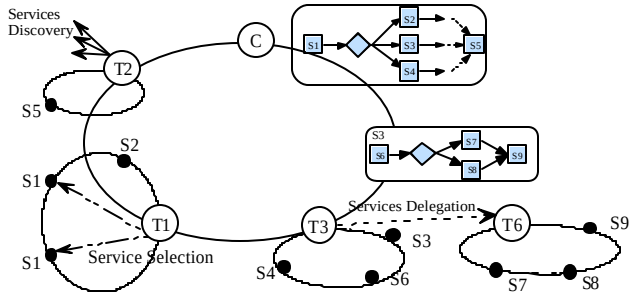

Figure 2: Proactive Collaboration

THE CAST-WS ARCHITECTURE

We have designed a team-based agent architecture CAST-WS (Collaborative Agents for Simulating Teamwork among Web Services) to realize our framework (see Figure 3). In this figure, the bottom box refers to Web service layer, and the top box depicts the detailed composition of CAST-WS, where the implemented team-based agent architecture (i.e. CAST) is extended with the *WS-Planning* and the *WS-Execution* parts for application to web service composition. In the following, we describe components of the architecture and explain their relationships.

The WS-Planning Component

The Planning component is responsible for composing services and forming teams. This component includes a service planner, a service discovery module, a team formation module, and an acquaintance model. Service discovery module is used by service planner to lookup in UDDI registry for required services. Team formation module, together with acquaintance model, is used to find team agents who can support the required services. A web service composition starts from user's request. The agent who gets the request is the composer agent who is in charge of fulfilling the request. Upon receiving a request, the

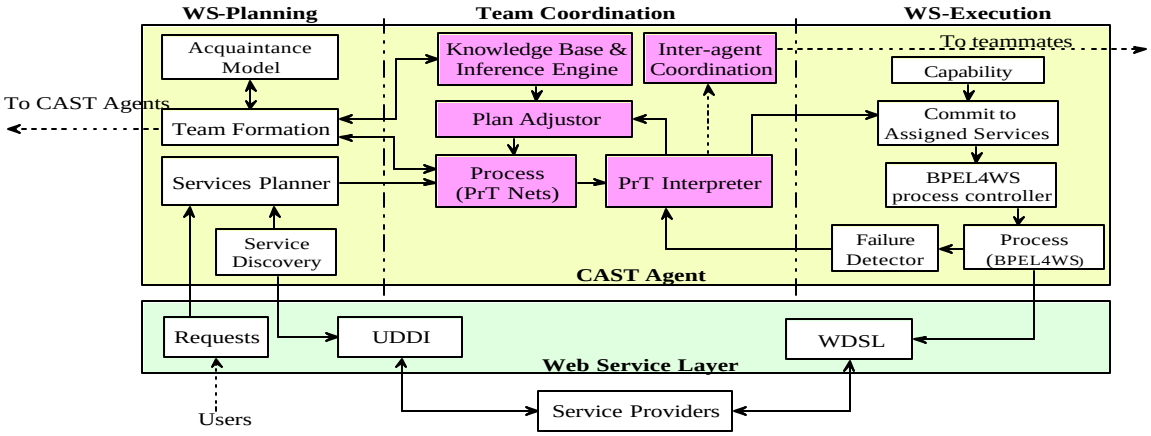

Figure 3: The Architecture of CAST-WS

composer agent turns the request into its persistent goal and invokes its service planner module to generate a business process for it. CAST agents use PrT nets to represent and monitor a business process.

The Team Coordination Component

A service manager agent uses the team coordination component to coordinate with other agents and execute the services. This component includes an inference engine with a built-in knowledge base, a process (in Petri-nets) shared by all team members, a PrT interpreter, a plan adjustor, and an inter-agent coordination module. Knowledge base holds the (accumulated) expertise needed for service composition. The inter-agent coordination module, embedded with team coordination strategies and conversation policies [Umapathy 2003], is used for behavior collaboration among teammates. Here we mainly focus on the process intepreter and the plan adaptor.

Each agent in a team uses its PrT net interpreter to interpret the business process generated by its service planner, monitor the progress of the shared process and takes its turn to perform those tasks dynamically assigned to it. If the assigned task is a web service, the agent invokes the service through its BPEL4WS process controller. If a task is assigned to more than one agent, the responsible agents need to coordinate their behavior (e.g., not compete for common resources) through the inter-agent coordination module. If an agent faces an unassigned task, it evaluates constrains associated with the task and tries to find a competent teammate for the task. If the assigned task is an abstract service (i.e. further decomposition required) and is beyond its capabilities, the agent treats it as an internal request, start composing a sub-process for the task and form another team to solve it.

The plan adjustor uses the knowledge base and inference engine to adjust and repair the process whenever an exception or a need for change in the process arises. The algorithm used by the plan adjustor utilizes the failure handling policy implemented in CAST. Due to the hierarchical organization of the team process, each CAST agent maintains a stack of active process and sub-processes. A sub-process returns the control to its parent process when its execution is completed. Failure handling is interleaved with (abstract) service executing: execute a service; check termination conditions; handle failures, and propagate failures to the parent process if needed. The algorithm captures four kinds of termination modes resulting from a service execution. The first results when the service is completed successfully. The second indicates that the process is terminated abnormally but the expected effects from the service has already been achieved "magically" (e.g. by proactive help from teammates). The third indicates that the process is not completed and is likely at an impasse. In this case, if the current service is just one alternative of a choice point, another alternative can be selected to re-attempt the service. Otherwise, the failure is propagated to the upper level. The fourth indicates that the process is terminated because the service has become irrelevant. This may happen if the goal or context changes. In this case, the irrelevance is propagated to the parent service, which checks its own relevance.

The WS-Execution Component

A service manager agent executes the primitive services (or a process of primitive services) through the WS-Execution component. The WS-Execution component consists of a commitment manager, a capability manager, a BPEL4WS process controller, an active process, and a failure detector. The capability manager maps services to known service providers. The commitment manager is used to schedule the services assigned to it in an appripriate order.

An agent ultimately needs to delegate those contracted services to appropriate service providers. The process controller generates a BPEL4WS process based on the WSDL of the selected service providers and the sequence indicated in the PrT process. The failure detector identifies execution failure by checking the termination conditions associated with services. If a termination condition has been reached, the failure detector throws an error and the plan adjustor module is invoked. If it's a service failure, the plan adjustor simply asks the agent to choose another service provider and re-attempt the service; if it's a process

failure (the unexpected changes make the process unworkable), the plan adjustor has to back-tracks the PrT process, tries to find another (sub-)process that would satisfy the task, and uses it to fix the one that failed.

DISCUSSION

The approach and architecture we have outlined has, at the core, a key element that distinguishes our efforts from many current efforts for web service composition. Instead of centralizing the process of web service composition, we have proposed to push the burden to the participants, the individual web services. A consequence of this shift in focus is that our approach allows us to interleave execution with planning. The framework and architecture we have outlined exhibits the following features, which provide distinct advantage over current web service composition approaches.

First, it supports an adaptive process suitable for the highly dynamic and distributed manner in which web services are deployed and used. With a clearly specified and shared joint goal, each agent commits to informing the team leader of any changes it may detect in the environment. Using this input, the composer agent is then able to make appropriate decisions at critical choice points. For example, an agent may proactively report an imminent service failure to the composer agent, who can decide on the appropriate response to the failure in a timely manner.

Second, it elicits a hierarchical methodology for process management. A complex process typically consists of several levels. A composer agent may not be able to decompose such a process to map its components to primitive services, either due to lack of knowledge or capability. Our framework allows a service composer to compose a process at a coarse level appropriate to its capability and knowledge, leaving further decomposition to competent teammates.

Third, it encourages separation of concern. Following the hierarchical management of the process, all agents share the tasks of composer agent e.g. execution, monitoring, and failure handling. For instance, to perform the service delegated by the leader, an agent may compose a lower-level process for it. The leader need not pay attention to the choice points, if there are any, in such a lower-level process; and may not even need to know the existence of these lower-level choice points. The team's distributed knowledge and specialized capabilities can thus be leveraged to offer better QoS.

Fourth, planning is interleaved with plan execution. Following this framework, an agent can act on a partial process (i.e., with some abstract services not being decomposed yet). For instance, while service *S1* is being executed by some service provider (delegated by *T1*), agent *T3* may still be deliberating on how to generate a process for service *S2*. Interleaving planning with plan execution can reduce the execution time, thereby improving overall efficiency of the system.

Our work in this direction has provided us with the fundamental insight that further progress in effective and efficient web service composition can be made by better understanding how *distributed* and *partial knowledge* about the availability and capabilities of web services, and the environment in which they are expected to operate, can be shared among the team of web services the must collaborate to perform the composed web service.

Interestingly, the pitfalls we anticipate in pursuing further work in this direction also stem from such *distributed* and *partial knowledge*. With the expectation that a large number of web services may be deployed and available on the web, which may be geographically dispersed, have different reliability quotients and may lead to unanticipated failures, a team-based model that requires proactive sharing of knowledge may be problematic. We expect that techniques from transaction management in distributed database settings may be useful in this regard.

Our planned work involves refining the architecture to clarify linkages to the underlying web-service technology stack, and developing mappings to these layers – with a view to implementing the CAST-WS architecture.

REFERENCES

1. Allen, Rob (2001). "Workflow: An Introduction". Workflow Management Coalition. Pg: 15-38.
2. Burg, Bernard (2001) "Agents in the World of Active Web-Services". Digital Cities. Pg: 343-356.
3. Oberleitner, Johann and Dustdar, Schahram (2003). "Workflow-Based Composition and Testing of Combined e-services and Components". Technical Report TUV-1841-2003-25, Vienna University of Technology, Austria.
4. Pires, Paulo; Benevides, Mario; and Mattoso, Marta (2002). "Building Reliable Web Services Compositions". Web, Web-Services, and Database Systems.
5. Smith, Reid G. (1980). "The contract net protocol: High-level communication and control in a distributed problem solver". IEEE Transactions on Computers, Vol: 29(12). Pg: 1104-1113.
6. Umapathy, Karthikeyan; Purao, Sandeep and Sugumaran, Vijayan (2003). "Facilitating Conversations among Web Services as Speech-act based Discourses". In Proceedings of the Workshop on Information Technologies and Systems (WITS 2003). Pg: 85-90.
7. Yen, John; Yin, Jianwen, Ioerger and et. al. (2001). "CAST: Collaborative Agents for Simulating Teamwork". International Joint Conference on Artificial Intelligence (IJCAI-01). Pg: 1135-1142.