HYBRID AGENT & COMPONENT-BASED MANAGEMENT OF BACKCHANNELS

M. Dragone, G. M. P. O'Hare
CLARITY Centre for Sensor Web Techologies, University College Dublin, Belfield, Dublin 4, Ireland

D. Lillis, R. W. Collier
School of Computer Science and Informatics, University College Dublin, Belfield, Dublin 4, Ireland

Keywords: Agent-oriented software engineering, Component based software engineering, Multi-agent systems.

Abstract: This paper describes the use of the SoSAA software framework to implement the hybrid management of communication channels (backchannels) across a distributed software system. SoSAA is a new integrated architectural solution enabling context-aware, open and adaptive software while preserving system modularity and promoting the re-use of existing component-based and agent-oriented frameworks and associated methodologies. In particular, we show how SoSAA can be used to orchestrate the adoption of network adapter components to bind functional components that are distributed across different component contexts. Both the performance of the different computational nodes involved and the efficiencies and faults in the underlying transport layers are taken into account when deciding which transport mechanisms to use.

1 INTRODUCTION

Agent-Oriented Software Engineering (AOSE) is viewed as a general-purpose paradigm advocating the engineering of complex distributed software applications as Multi Agent Systems (MASs), i.e. loosely-coupled, autonomous, situated and social agents. For example, agents can be instrumental in confronting the high degree of variability and dynamicity in modern pervasive computing applications.

The idea that MAS features, such as goal-based reasoning, standardized agent communication languages (ACL) and associated protocols, can support flexible and adaptive management of system distribution over multiple and potentially heterogeneous computational nodes, was first proposed for RETSINA (Sycara et al., 2003), whose hybrid communication style provides ACL-level, implementation-agnostic specifications of the coordination activities necessary to orchestrate secondary (non-ACL) communication channels (backchannels).

To aid the development of systems that address context-aware, open and adaptive software while preserving system modularity and promoting the re-use of existing frameworks and methodologies, we have proposed the Socially Situated Agent Architecture (SoSAA) (Dragone, 2007; Dragone et al., 2009).

SoSAA builds on both Component Based Software Engineering (CBSE) and AOSE to provide a complete construction process with its associated software framework. The key focus is on the interaction between the component and agent layers of the framework. In this paper we briefly summarise the most important characteristics of SoSAA and describe its application to ground RETSINA-style specifications by providing a set of concrete operators and by defining a simple but extendable service for backchannel management.

2 AGENT-BASED BACKCHANNEL MANAGEMENT

The ability to combine different communication mechanisms is an important feature requirement in order to support system's adaptation in modern distributed and heterogeneous software systems. Often, the differences between the type of data exchanged within a single application (e.g. in terms of content, rates and bandwidth requirements) cannot be

Dragone M., M. P. O'Hare G., Lillis D. and W. Collier R. (2009).
HYBRID AGENT & COMPONENT-BASED MANAGEMENT OF BACKCHANNELS.
In *Proceedings of the 4th International Conference on Software and Data Technologies*, pages 153-158
DOI: 10.5220/0002259901530158

handled by a single toolkit or communication mechanism. For example, one system may need to combine CORBA communication mechanisms with middleware that specifically addresses the distribution of media data (e.g. video/sound), or combine data discovery mechanisms, e.g. based upon data dissemination techniques such as multicast, with point-to-point agent communication systems. All these communication mechanisms provide dedicated and optimum solutions to specific problems or provide access to a large base of associated tools and libraries and should be integrated for opportunistically satisfying all the systems needs and constraints.

Supporting ubiquitous and mobile systems adds an element of dynamicity that is also difficult to handle. Contrary to static and offline integration scenarios, where all the communication links can be preconfigured, in open / ubiquitous environments, the different elements are dynamically distributed. This makes it vital to be able to alter communication pathways at run-time, e.g. by contacting different servers while the user moves within a ubiquitous infrastructure or interacting on-demand through portable and/or wearable devices that can be characterized by different computational, network and presentation capabilities. In these cases using multiple communication mechanisms may be instrumental in guaranteeing the adaptability of the system to such an environment.

However, in addition to the complexity of managing distinct data-exchange formats, combining multiple communication mechanisms also adds the overhead of managing their their relationships, for example, to guarantee some global policies, such as policies for security or resource management. For instance, it might be appropriate for all the mechanisms employed to respect some high-level run-time policy dictated by a specific application. For example, in tele-conferencing systems, session termination may require the closure of all the single connections that were established in the process. In general, limited resources in terms of memory, network bandwidth, CPU, etc, often posits a conservative approach consisting of shutting down unused modules and generally releasing resources after use is usually preferable. Conversely, if one data connection carrying one type of data terminates prematurely, e.g. for a problem isolated to a specific transport layer, it should be possible to detect the condition and re-establish the communication, e.g. to restore a session, or to support more sophisticated error-recovery procedures.

Such an approach is advocated in the RETSINA architecture. RETSINA supports the specification (in an agent communication language such as KQML or FIPA-ACL) of the conversational policies agents should adhere to in their interactions and the relevant ontologies for achieving semantic interoperability. Crucially, the ACL-level is used to coordinate multiple heterogeneous communication systems in order to bind together different functional components and support agent-based applications running under a wide variety of conditions, e.g. different operating systems, devices and programming languages. This approach results in a two tiered communication strategy, in which low-level communications, called backchannels, are represented and coordinated in the ACL level. However, RETSINA provides only ACL-level, implementation-agnostic specifications which can lead to the development of such features.

3 SoSAA

SoSAA incorporates modularity by applying the principles of hybrid agent control architectures. Popularised by their use in robotics (e.g. in (Gat, 1992)), hybrid control architectures are layered architectures combining low-level behaviour-based systems with high-level, deliberative reasoning apparatus. From a control perspective, this enables the delegation of many of the details of the agent's control to the behaviour system, which closely monitors the agent's sensory-motor apparatus without symbolic reasoning.

The original solution advocated by SoSAA is to apply a hybrid integration strategy to the system's infrastructure, as illustrated in Fig. 1. SoSAA combines a low-level component-based framework with a MAS-based high-level framework. The low-level framework operates by imposing clear boundaries between architectural modules (the components) and guiding the developers in assembling these components into a system architecture. It then provides a run-time computational environment to the high-level framework, which then augments it with its multi-agent organisation, ACL-level interaction, and goal-oriented reasoning capabilities.

The fundamental motivation behind such an approach is that application agents in SoSAA do not just collaborate to carry out the different objectives of the system, e.g. championing different sub-goals by attending to different functional requirements. Each agent is first of all a member of an agent community inhabiting the same computational environment. As such, it may be required to compete with the other agents for the computational resources in the system. Under this perspective, SoSAA provides a platform from which these agents can access the system's resources (e.g. hardware, CPU, data, network), coordinate their activities and resolve possible conflicts aris-

ing at this level.

A number of widely-used mechanisms in the CBSE community are supported by SoSAA's low-level framework:

- Support for different component collaboration styles, such as event notification, data messaging or procedural calls (services).
- Reflection features, which can be used to define specific component collaborations in terms of provided (outgoing) and requested (ingoing) interfaces.
- Container-type functionalities, used to load, unload, activate, de-activate, configure, and query the set of functional components loaded in the system.
- Brokering services, which can be used as late-binding mechanisms by the components to locate suitable collaboration partners at run-time for every collaboration style supported by the framework.
- Binding operations, with which the client-side interfaces (e.g. event listeners, data consumers, service clients) of one component can be programmatically bounded to server-side interfaces (e.g. event sources, data producers, service providers) of other components.
- Provision of framework-wide mechanisms, such as logging, life-cycle management, and scheduling/control injection of process-type components.

Additionally, the high-level framework employs meta-level sensors and effectors that collectively define an interface, namely the *SoSAA Adapter Service*, that can be used to sense and act upon elements and mechanisms of the low-level framework by loading, unloading, configuring and binding components.

The other mechanism adopted for implementing the SoSAA hybrid strategy is a callback design pattern with which dedicated SoSAA component agents can collaborate with the low-level framework by monitoring it and by registering themselves as controllers for specific events. In this manner, these agents are then able to override the default behaviour of the mechanisms provided by the low-level infrastructure, and thus take more informed, context-sensitive decisions. Crucially, while this can be done by taking into consideration application-specific situations, as both infrastructure and application-level issues are equally addressed at the ACL-level, these two levels can now be confronted by different agents. Instead of overloading one agent with both types of concerns, such an approach enables the definition of distinct agents

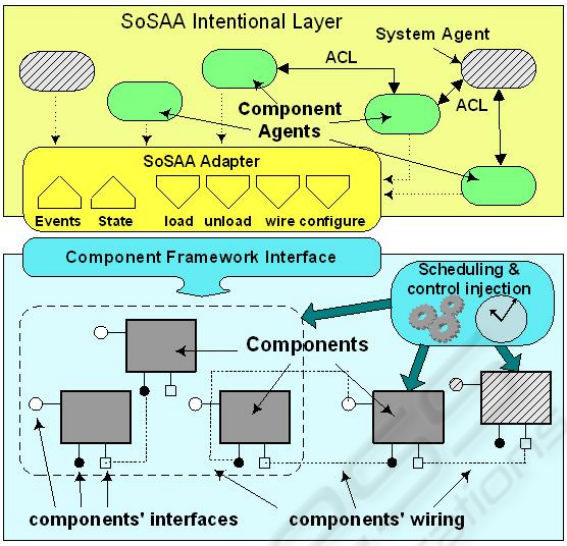

Figure 1: SoSAA Hybrid Integration Strategy.

that focus only on certain aspects and that address conflicts and inter-dependencies at the ACL-level.

Figure 1 illustrates this point by showing the resulting multiagent organization in a typical SoSAA node. Specifically, the low-level framework provides the interface for operating both at the application and at the infrastructure level. Accordingly, depending on their interests, SoSAA component agents can be distinguished between application and infrastructure agents. Such a clear separation is fundamental for promoting not only the efficiency and the portability of the resulting systems, but also for separating the different concerns at design time in order to facilitate the adoption of a modular development process.

From an integration perspective, the SoSAA hybrid approach favors the distinct evolution of the software systems it combines, thus more adequately satisfying the requirements for extensibility demanded of the overall framework. Since the adapter service is defined in terms of both standard agent capabilities and a core component model, SoSAA's design facilitates the use of different agent toolkits and component frameworks. While the former can be programmed according to different cognitive models, domain and application-specific issues can be accounted for in the design of the underlying functional components.

The current instantiation of SoSAA is based on two open-source toolkits, namely: (1) Agent Factory (AF) (Collier et al., 2003) is a modular, extensible, open source framework for the development of FIPA-compliant Multi Agent Systems whose constituent agents are implemented in a purpose-built Agent-Oriented Programming language known as AFAPL that supports programming of agents in

terms of beliefs, commitments and goals (Collier and O'Hare, 2009); and (2) the Java Modular Component Framework (JMCF), a java-based library developed in UCD and used to integrate SoSAA with different component-enabling technologies, such as Java Beans and OSGi. JMCF comes with a package of built-in component types and base-class implementations, each providing (i) component-container functionalities (e.g. loading/unloading/configuration of components), (ii) brokering functionalities to register and find component services, and (iii) support for the runtime execution of active (process-type) components. Further details on JMCF can be found in (Dragone et al., 2009).

4 SoSAA HYBRID COMMUNICATION

It is relatively easy to satisfy the requirements set on the SoSAA low-level framework for a single platform, as brokering and container functions can use inter-process (e.g. memory sharing) communication.

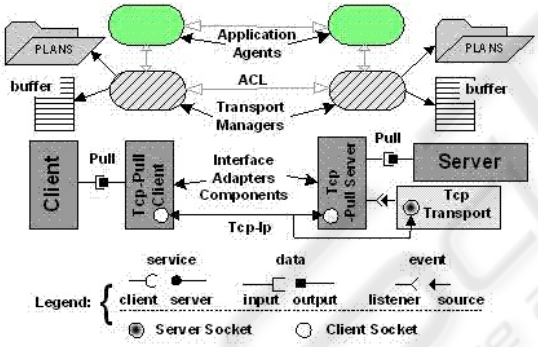

Figure 2: SoSAA Backchannel Management.

Figure 2 helps to illustrate the realisation of the hybrid backchannel management in SoSAA. Interface adapter components provide the bridge between the standard interface classes used for inter-process component collaboration and the backchannels used to connect the corresponding components' interfaces across the network. In the example depicted in Figure 2, two components (*client* and *server*) are remotely connected through a Pull data interface. Local connectivity is resolved by simply assigning, on the client-side, a reference to a server-side implementation of the Pull interface, which is defined as a single *pull()* method returning a Java *Object*.

A remote connection between these components is achieved by interposing a TcpPullServer, which is bound to the server component exporting the Pull interface, and a TcpPullClient component, which is bound to the client component requiring it. Figure 2 also shows how the control of the backchannel is shared among application component agents and system component agents (called *transport managers*) in the respective nodes. Specifically, whenever the client application agent wants one of its components (the client component in the example) to start exchanging data with a remote one, it will issue an ACL request to the remote agent requesting it to find a suitable partner (the server component in the example).

In general, application agents will use application-specific protocols (e.g. auction based) to agree on a collaboration between their respective functional components before delegating the details of the connection to their respective transport managers (see Figure 3). To reduce the need to negotiate specific transport mechanisms for each collaboration, each transport manager uses a *heartbeat protocol* to introduce themselves to each other. This protocol is also used to exchange information concerning the actual transport mechanisms supported by each node, together with the relevant (transport-specific) details required to open new connections (e.g. the port for an TCPIP backchannel).

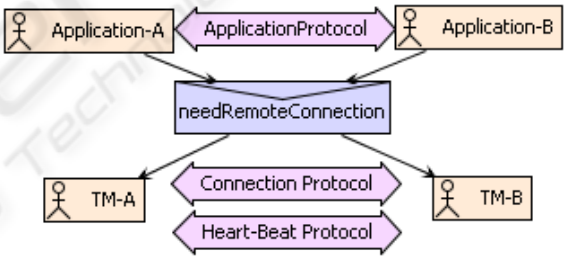

Figure 3: Interaction Diagram, Application and Transport Manager Agents.

In order to trade-off the impact of this communication with the scalability and the ability of the system to adapt to dynamic resources, due for instance, to failures in the underlining transport layers, this information is then cached by each transport manager. Each entry in the cache is also supplemented with usage statistics, including the total number of interfaces established toward each collaborating node, their latest throughput, and the last time they were successfully (or unsuccessfully) used. Collectively, this information enables the definition of transport selection strategies addressing QoS optimization and system's adaptation to changing environmental conditions.

The final selection of the respective transport adapters, their configuration and their wiring to the local functional components is coordinated between the two transport managers through the use of a connec-

tion dialogue plan called `setupRemoteConnection`. Each transport manager activates a copy of this plan, partially illustrated in Figure 4, as soon as an application agent requests the creation of a connection between a local and a remote component. Each instance of this plan is then used to keep track of all the intermediate steps necessary to create the desired connection, by sending requests to the remote transport manager, and by mapping incoming requests and responses to a series of states, captured by sub-goals and sub-plans that correspond to the dialogues state.

Initially, each transport manager queries (through the SoSAA Adapter) its local components to find out if the local interface is a client- or a server-side interface and also to find out the associated Java interface class (TcpPull in the example in Figure /refFig2). In the first case, the transport manager adopts the goal `GOAL(clientConnection(<terms>...)` while in the second case the transport manager adopts the goal `GOAL(serverConnection(<terms>...)`. In both cases, the terms posted with these goals specify the name of the components and their interfaces and also the Java interface class. However, they leave open any details concerning the specific protocol and the adapter to be used. These are represented in the goal as variables to indicate that any possible instantiation will suffice to satisfy the goal. Further, these goals are posted as MAINTAIN goals.

Once their role in the new connection has been established, each transport manager adopts different sub-plans to carry out that role. In the example depicted in Figure 2, the left-hand agent takes charge of the client-side connection by driving the coordination with the the other transport manager. First of all, the `setupClientConnection` plan is exploited, together with the AF plan-selection mechanism, to finally identify a suitable transport mechanism. The postcondition of this plan matches `GOAL(clientConnection(...))` while its precondition checks that both platforms can currently handle the same transport protocol (without specifying it). As such, upon posting the goal the AF reasoning engine automatically generates a number of possible options, all committing to the activation of the same setupClientConnection plan but with different transport protocols specified in their activation arguments.

Conversely, the transport manager in charge of the server-side interface will activate the `setupRemoteConnection` plan, and inform the client-side manager that it is ready to setup the connection. Once this message is received, the connection protocol highlighted in figure 4 commences. The client-side requests the server-side manager to setup a server adapter for the transport protocol selected. If nothing has changed since transmitting its last heartbeat update, the server-side manager will install the server-side adapter and inform the client-side manager of the details needed to contact it. Alternatively, if the given transport protocol is not available any longer on the server-side, the server-side will reject the request. This will cause the failure of the client-side plan and the automatic activation of the next available option. Finally, the actual connection creation is delegated to wiring plans `setupClientAdapter` and `setupServerAdapter` that account for the characteristics of the transport protocol selected in the first negotiation phase.

Noticeably, the standardized interface with the component layer, which supports loading/configuring and binding of components, the wiring plans can use declarative descriptions of the different wiring steps rather than the steps being hard-coded in different sub-plans. This makes much easier adding the support for new transport protocols, and knowledge of these wiring steps can be acquired by the agent at run-time, or even exchanged through ACL messages among different agents. In general, this is a crucial feature awarded by the SoSAA design which makes the agent system capable of acquiring new capabilities at runtime and in line with dynamic binding mechanisms exploited in the specific component layer.

In the case of the TcpPull connection depicted in Figure 2, the `setupClientAdapter` plan will: (i) load the TcpPullClient component, (ii) bind it with the local client component, and (iii) configure it with the address of the server's node and the name of the remote server-side component. On the server side, the `setupServerAdapter` plan must: (i) load a TcpPullServer, and (ii) bind it to the server component. In the current implementation, the actual initialization of the connection in the server side is performed by a TcpTransport service component, also depicted in Figure 2, which accepts all incoming requests on a single server socket and notifies the required TcpPullServer component with the resulting client socket. In each case, the wiring plan records the details of the connection by storing a `Belief(client/serverConnection(...))` matching the goal posted at the beginning of the connection dialogue plan.

This hierarchical organization between application agents, transport managers, and components, is designed to handle functional and non-functional requirements in terms of goals with decreasing levels of abstractions. Application agents are driven by higher-level functional goals (e.g. "*componentA* needs data from *componentB*"), which are resolved with the selection of suitable collaborating components. Trans-

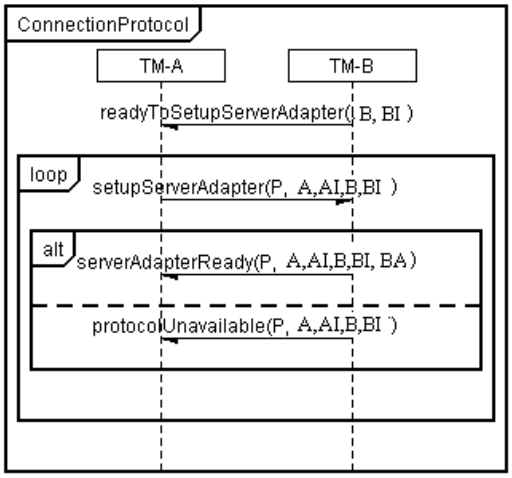

Figure 4: AUML TransportManager's dialogue plan (setupRemoteConnection) Legend: P: Communication Protocol;A: Name of component A; AI: Name of interface for component A; B: Name of component B; BI: Name of interface for component B; BA: Name of network adapter for component B.

port agents are driven by lower-level goals concerning the initiation and the maintenance of concrete inter-component connections. They are also responsible for reacting in case of failure and for trying alternative transport mechanisms before notifying the application level of the connection error. The component layer is passive and carries out higher-level instructions and performs the default behaviour hard-coded in the functional components (e.g. "always provide data to a pull request") or system components (e.g. "always accept an incoming TCPIP connection").

By monitoring the adapter components, through the `focus` operator, the SoSAA Adapter is aware of the state of the underlining communications, which also means that there is no need to exchange any more messages between the transport manager, other than the minimum three messages necessary to synchronize and agree on the terms of the communication.

Errors in the transport protocols or unexpected exceptions, such the closing of a socket, are simply reported to the local transport manager in form of `Belief(error(?Adapter, ?ErrorDetails))`. By dropping `Belief(client/serverConnection)` from the agent's beliefs, the AF reasoning engine automatically tries to restore the connection, by reactivating the connection dialogue plan.

Finally, the SoSAA Adapter can be used to define bottom-up connection strategies, whose handling connections that are initiated within the component-layer. In the case of a TCPIP connection, the transport manager can simply intercept the initiation of a new connection (by overriding the event notification between the TcpTransport and the TcpPullServer component) and collaborate with one or more application-level agents to take a more informed decision upon the connection, e.g. in order to implement load-balancing or application-specific security mechanisms.

5 CONCLUSIONS

This paper presents an approach to backchannel management built using the Agent Factory and JMCF implementation of SoSAA. Further, it attempts to show how both SoSAA and the backchannel management mechanism can be used to construct hybrid agent-component applications. A key advantage of our approach arises from the clear separation of concerns that exists between the application logic agents and transport manager agents that form part of the SoSAA infrastructure, and which manage the creation, configuration, and maintenance of backchannels.

REFERENCES

Collier, R., O'Hare, G., Lowen, T., and Rooney, C. (2003). Beyond Prototyping in the Factory of Agents. *Multi-Agent Systems and Application III: 3rd International Central and Eastern European Conference on Multi-Agent Systems, Ceemas 2003, Prague, Czech Republic, June 16-18, 2003: Proceedings*.

Collier, R. W. and O'Hare, G. (2009). *Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches*, chapter Modeling and Programming with Commitment Rules in Agent Factory. IGI Publishing.

Dragone, M. (2007). *An agent-based robot software framework*. PhD thesis, School of Computer Science and Informatics, University College Dublin.

Dragone, M., Lillis, D., Collier, R. W., and O'Hare, G. M. P. (2009). SoSAA: A Framework for Integrating Agents & Components. In *Proceedings of the 24th Annual Symposium on Applied Computing (ACM SAC 2009), Special Track on Agent-Oriented Programming, Systems, Languages, and Applications*, Honolulu, Hawaii, USA.

Gat, E. (1992). Atlantis: Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 809–815.

Sycara, K., Paolucci, M., Van Velsen, M., and Giampapa, J. (2003). The retsina mas infrastructure. *Autonomous agents and multi-agent systems*, 7(1):29–48.