CHANNEL-BASED ARCHITECTURE FOR DYNAMICALLY RECONFIGURABLE NETWORKS

Jeroen Valk [a] Jan Peter Larsen [a] Peet van Tooren [a]
Adriaan ter Mors [a]

[a] Almende B.V., Westerstraat 50 3016 DJ Rotterdam

Abstract

Agent technology is often suggested as a tool for developing software that is more adaptive in the face of changes in its environment. In agent-based approaches, complex automated systems are built from components that communicate with each other. To enable adaptive behaviour, both the components themselves and the communication architecture should be easily changeable. This paper presents the common hybrid agent platform (CHAP)[1] that enables dynamic reconfiguration of networks composed of small agents that may be changing themselves.

1 Introduction

Lack of adaptivity in automated systems has been a serious problem in the software industry over the years. Typical industrial applications consist of many lines of code which are difficult to understand. When a new project is started, and software is developed from scratch, knowledge about the datastructures and control flow is sufficient to build a working program. During the lifetime of the program, however, in-depth knowledge about the software may fade away rapidly. Particularly, in a changing environment, the demand for software changes may be high. To fulfill this demand, developers are often faced with a difficult (if not impossible) task: to change incomprehensible lines of codes.

One way to resolve the lack of adaptivity in software is to build systems that can more or less automatically adapt based on evaluation of its behavior with respect to the software's intentions. Information that is useful in this respect might be, user preferences, performance indicators, or feedback. Many of these approaches focus on automatic reconfiguration without human intervention. For example, in a DARPA Broad Agency Announcement, self-adaptive software is roughly defined as follows: "Self-adaptive software evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible." The idea here is that the ability to adapt is programmed into the system and will manifest itself at runtime without any human intervention.

The adaptivity problem in software allows an alternative approach as well: to implement software in such a way that only little knowledge about the complete system is required to make small changes in its functionality. This is the idea of compositionality, which has been exploited in many programming paradigms. An interesting paradigm in this respect is that of agent-oriented programming [4]. In this newly emerging paradigm, components are no longer static, but they have an ongoing interaction with the environment. The idea is that systems based on a dynamically reconfigurable network of components are easier to comprehend and adapt locally. In existing software, however, components are usually of a monolithic nature, which cannot be understood in terms of interactions of simple functionalities. We therefore aim to (re)implement software as a dynamically reconfigurable network of simple components and check whether this software is indeed more adaptive.

[1] A demo of this system will be shown at the conference.

Traditional software development techniques and industry standards to connect components include streams, (secure) sockets, or HTTP; these are all standardized ways to let independent components (called threads or processes) communicate over so-called *communication channels* which consist of two endpoint called connectors which are linked in pairs: data that is written on a connector can be read (by another component) from the other endpoint of the channel. However, existing industrial standards provide little or no support for dynamic reconfiguration. Sockets, for example, are typically designed for building pre-wired applications with hardly any support for the dynamic reconfiguration of the communication infrastructure.

A more promising area for our experiments can be found in the realm of formal methods and verification techniques. A variety of formalism has been developed: e.g., temporal logic, process algebra, MAUD, CREOL, REO [2, 3, 1]. In the face of dynamically reconfigurable networks, REO is an interesting framework, because its key concept is that of a so-called *mobile channel*. A mobile channel is a communication channel that can be created and destroyed, and connectors can be passed on from one component to another. All these channel manipulation operations can be performed while the system is running without affecting the integrity of the collective system state. Unfortunately, however, a development and execution environment for REO is still missing. To bridge this gap, we suggest to use an architecture for channel manipulation that has been developed at Almende. The idea is to use channel manipulation offered by the architecture as a basis, and to build REO primitives, i.e., nodes and channels, on top of this architecture.

The outline of this paper is as follows. First, we will briefly introduce the REO framework. Next, we will present our architecture and suggest how it could be useful for REO. Next, we will describe some applications for which the architecture could provide solutions. We conclude the paper with suggestions for future work on the architecture.

2 The REO framework

In [1], Arbab et al. suggest a formal language for the exogeneous coordination of components via so-called *mobile channels*. In this channel-based language, a component is conceived as a flow of control which communicates with its environment (which just consists of other components in the system) using communication channels. Communication channels consist of two endpoints called *connectors* which are linked in pairs. Data that is written on a connector can be read (by another component) from the other endpoint of the channel.

An important application of mobile channels is automated configuration of a communication network. For example, we can have three components A, B, and C where A decides (without human intervention) that B and C have to communicate with each other. Then A can create a channel and pass one connector to B and the other connector to C. Existing communication standards do not provide support for building components which manipulate communication channels in the way described above.

Another interesting feature of REO is that it considers channels in a very general sense. That is, REO does not specify the behavior of channels, but only the connectivity. A REO circuit is just a kind of *directed* graph with nodes and arcs; only the direction of arcs is a bit peculiar. That is, besides the normal directed arcs that run from one endpoint to the other, we have (i) arcs that run from both endpoints into the "nothing", and (ii) arcs that run from the "nothing" into both endpoints. In a graph, we can distinguish three kinds of nodes: (i) nodes that have only incoming arcs, (ii) nodes that have only outgoing arcs, and (iii) nodes that have both. In REO,

(i) a node with only *incoming* arcs is called an *output node*;

(ii) a node with only *outgoing* arcs is called an *input node*;

(iii) all other nodes are called *mixed nodes*.

Only the behavior of nodes is fixed in REO, but channels are allowed to vary widely. Probably, any kind of useful channel can be defined in REO and all these channels can be used within a single circuit. This allows us to build very rich and powerful circuits. For example, it allows the formal specification of circuits that consist of both synchronous and asynchronous channels. Most existing formalisms usually choose either one of these channel types.

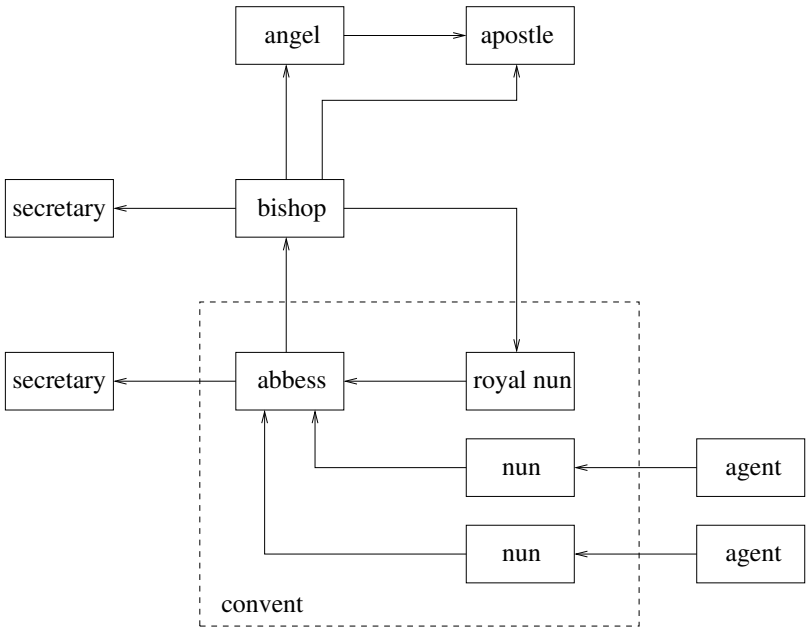

Figure 1: a simple monastery.

We have now discussed the static characteristics of a REO circuit. In the next section, we will describe how to build up and change such circuits in a distributed and dynamic way using our channel manipulation architecture.

3 Architecture for Channel Manipulation

The basis of the channel-manipulation architecture which we suggest is an agent-based platform called CHAP (Common Hybrid Agent Platform). In CHAP, channel-manipulation primitives are offered in a distributed way using a large number of behavioral components[2]. These components for channel manipulation are called *nuns*[3]. A nun provides a kind of common interface that allows other components to create, transfer, and hangup communication channels. In this way, communication is possible without having to know any hardware specific details about the communication technology that is used. A nun can offer channel-manipulation functionality to any component in CHAP, but it can only service one component at a time. The functionality is offered by means of messages passed on a communication channel between the nun and the component that is being served. Note that this is similar to a traditional client/server model. The difference is that in CHAP there is a very large number of clients and servers (nuns) that operate concurrently.

To coordinate inter-nun communication, collections of nuns are under the supervision of a so-called *abbess*. Collectively, an abbess together with all her nuns is called a *convent*. The abbess registers the nuns in the convent and makes sure the nuns can find each other when needed. Additionally, the abess provides an interface between the nuns and more low level components in CHAP which offer CPU and communication management. In this respect, CHAP is a layered architure where the convent is build on top of a lower layer called a *religious order*. In turn, agent-based applications that use channel manipulation can be built on top of the convent layer.

In the current version of CHAP, a (religious) order consists of three components: (i) a *bishop* that interfaces with the abbess and nuns, (ii) an *angel* that is responsible for creating new threads of execution, and (iii) an *apostle* that creates communication channels. An order together with a convent on top of it is called a *monastery*. Figure 1 shows the most recent version of a monastery used in CHAP where the components are structured hierarchically.

[2]In CHAP, each behavioral component is represented by a single thread of execution.

[3]As a methaphor, we use names of members of a religious order to identify the various roles of the components within CHAP.

At startup, the CHAP architecture bootstraps from the angel. That is, to start CHAP just one angel has to be started. The angel may then may start one or more monasteries on a machine where no CHAP components are running yet. A typical angel would be responsible for starting two types of components: bishops and apostles. Apostles are the providers of communication channels where different apostles could provide different types of communication channels. Bishops are responsible for deploying one or more abbesses which in turn found so-called convents consisting of nuns. Currently, we use a very simple lower layer where the angel starts one apostle and one bishop, and the bishop deploys one abbess. The bishop can best be viewed as a kind of server that can handle requests for component and channel creation. The bishop does not handle these requests itself, but (i) uses the angel to spawn new threads whenever a request for component creation arrives, and (ii) uses the apostle to create a new channels upon request.

The convent not only serves the components in the application in application layers on top of the monastery, but also the bishop in the first layer of the monastery. The bishop is served by what we call a Royal nun which offers the same interface as the other nuns in the convent. The bishop uses the Royal nun to bootstrap the agent application.

Nuns extend the functionality of the bishop: i.e., they can be used to create new agents and to request communication channels. Additionally, the nuns can be used to connect components with each other. To do so, components must first be registered in the convent by means of a register request to a nun. The nun then sends back an identifier which can be used to connect components with each other. For example, consider the simple application with three agents A, B, and C where A wants to connect B with C. If A creates the components B and C itself then it could first register B and C at the convent. Component A can do this by sending two register request to its own nun to get two identifiers. Then it can send these two identifiers in a connect request to its own nun. Upon registration, components B and C have been assigned a separate nun in the convent and they can use these nuns to inquire for incoming connectors. This inquiry will provide B and C with the two endpoints of a channel, because of the connect request issued by component A.

In a typical agent-based application, more that one monastery may be involved. These monasteries can be running on different machines which are connected by a physical network. Within a monastery, the communication channels are homogenous. The communication channels of different monasteries may be of a completely different type however. This allows the application to use different communication technologies in a transparent way. To use different types of channels the components should register at multiple convents. This means that in a typical agent-based application several nuns may serve a single component.

To provide an idea of how CHAP works in detail, let us give an illustration of a communication flow in the monastery. In the simple example we discussed earlier, agent A first creates and registers components B and C. Figure 2 shows the request that will be exchanged if agent A (the bootstrap agent) creates a new component and registers this component at the convent. First, the bootstrap agent creates a new agent by sending an agent request to its nun. The nun cannot handle this request so it delegates the request to the abbess. The abbess, in turn, delegates the request to the bishop. The result of the two subsequent delegate requests is that the agent talks directly with the bishop. The bishop gets an agent type from the bootstrap agent, sends a channel request to obtain a communication channel from the apostle, and sends a baptize request to create an agent of the specified type. Note that one endpoint of the channel received from the apostle is used by the angel to make an initial connection to the newly created agent. Next, the bootstrap agent issues a register request to register the new component at the convent. Along with the register request the bootstrap agent sends the connector to its nun to which the newly created agent is connected. The nun then gets a new identifier for the agent by sending a *getid* request to the abbess. The identifier obtained from the abbess is sent to the bootstrap agent. Finally, the nun creates a new nun and informs this new new nun about its identifier and the connector on which this new nun should listen.

4 Applications

We identify several domains were adaptivity in software based on dynamic reconfigurable communication between components can have far-reaching consequences. A basic version of CHAP has already been commercially implemented in some industrial sectors. Scheduling, matching, and call

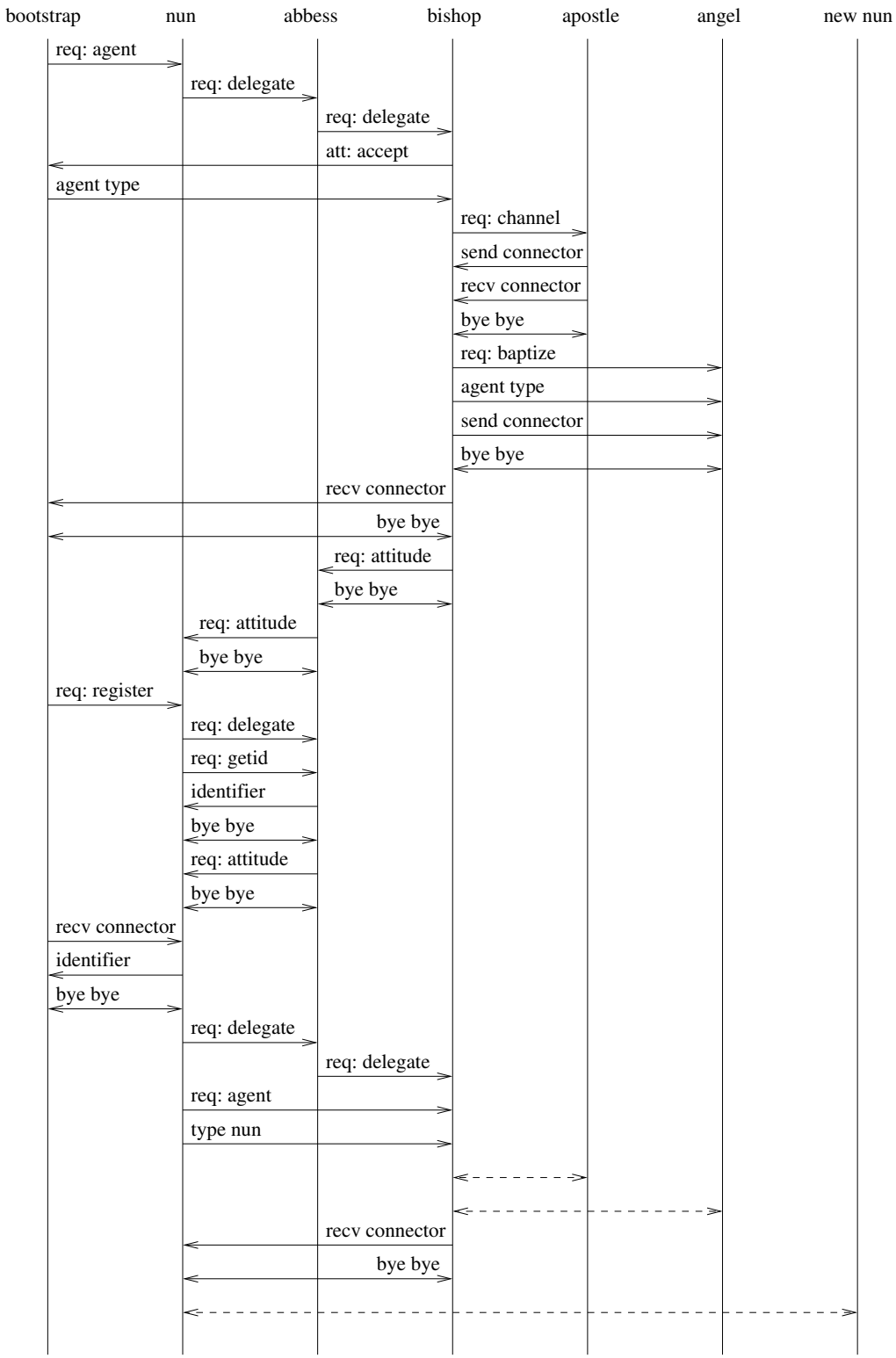

Figure 2: message flow generated by the bootstrap agent.

routing are often the main functionalities of the provided solutions. These implementations focus on connecting humans through adaptive matching mechanisms. Every human user is represented by an agent that can be told or even learn (through feedback) the user's individual preferences. These system are used to solve communication bottlenecks which occur when organizations need to communicate very last-minute with a lot of people, e.g. in (human) resource planning, or in knowledge management.

In these implementations, it is mainly the components themselves that change their behavior according to the needs of the environment. The full potential of dynamic reconfigurable networks becomes clear when very large numbers of small components make up the system. In these situations, only through communication can the system adapt to its designed functionality. A typical domain where our proposed architecture could be essential is within complex logistic planning processes, especially when the solution has to be found through distributed communication processes between many heterogeneous actors.

Concurrent development of applications for logistical purposes has started by using the CHAP architecture to develop an agent-based solution to tackle large-scale complex logistical processes. Potential application areas that have already been identified are multi-modal transportation, and ground handling in airports (multi-actor networks). A distributed model of stakeholders at a micro level (packages, trucks, etc.) and the organization of communication between them offers possibilities to find incremental solutions for last-minute incidents.

CHAP offers a perfect architecture to develop such a distributed network of actors. For a feasible industrial agent-based implementation, however, large-scale reliable solutions are required due to the vast amount of interacting agents needed for these complex coordination processes. The same is true for many other potential application areas. Further development of CHAP will therefore focus mainly on robustness and scalability.

5 Future Work

We identified the following of attributes which are important as future work.

Trust and Consensus. In a system where all components can be trusted , e.g. because all components have been developed centrally, the effects of interactions will be as intended by the designer. But if third parties are allowed to change components or add components to the system, consensus with regard to the meaning of a message may no longer be guaranteed. This triggers a need for agents which can deal effectively with undefined semantics. For example, think of agents that delegate requests to others which they don t understand.

Robustness and Redundancy. To obtain a robust system that can achieve certain performance guarantees, we think of techniques that provide redundancy (i.e., having more components capable of delivering the same functionality).

Failure detection. Even in a system where all components can understand each other perfectly, faulty behavior may be the result of physical limitations of the hardware on which the software is run. In most automated systems, physical limitations are often detected by a human operator. The CHAP architure must be able to detect automatically if undesired system behavior is due to lack of resources and take appropriate actions. These actions could be (i) to relocate processes on the physical hardware, or (ii) to limit resource consumption if agents can function appropriately without these resources.

References

[1] Farhad Arbab. Abstract behavior types: a foundation model for components and their composition. *Sci. Comput. Program.*, 55(1-3):3–52, 2005.

[2] J. A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 1984.

[3] Einar Broch Johnsen and Olaf Owe. An asynchronous communication model for distributed concurrent objects. In *Proc. 2nd Intl. Conf. on Software Engineering and Formal Methods (SEFM'04)*, pages 188–197. IEEE Computer Society Press, September 2004.

[4] Yoav Shoham. Agent-oriented programming. *Artificial Intelligence*, 1993.