STITCH: Middleware for Ubiquitous Applications

Damián Arregui, Christer Fernström, François Pacull, Gilbert Rondeau and Jutta Willamowski

Xerox Research Centre Europe, Meylan, FRANCE
Firstname.Lastname@xrce.xerox.com

Abstract

In this paper we present STITCH, a middleware that facilitates the development of ubiquitous user-centric and context-aware applications. These systems are built from distributed components which are highly heterogeneous, covering a large range of hardware configurations, from sensors to computers, and with software ranging from embedded software with small footprint to larger scale services. We believe it is essential to provide infrastructure support to express system behaviors across such components. Our approach relies on an integrated view of distributed events and transactional resource manipulation. The former is well-suited to loosely couple small devices, while the latter allows for verification of distributed conditions and coordination of actions. STITCH provides a common layer of abstraction allowing application developers to take advantage of both paradigms.

1. Introduction

From the appearance of the first computers to the advent of the global networks, a number of new technologies progressively made their way into our daily lives. Designers and developers packaged them into applications that were supposed to improve our efficiency either at producing value or at entertaining ourselves. A number of them are widely successful: office applications, databases, the World-Wide Web, e-mail, games, electronic payment, etc. We have become accustomed to interact with such tools at different stages, switching back and forth from real-world activities to the virtual world of software applications.

However, as technology continues to evolve, more and more computing and networking capabilities are squeezed into small hardware devices (PDAs, mobile phones, smart cards, sensors, RFID tags, etc.), hence our expectations grow: the physical and virtual worlds should be more tightly integrated and the effort in switching from one to the other should be minimized. We would like to focus on the activities we perform rather than on the computer systems that facilitate them and we will no longer accept to sit down at a desk in order to benefit from computer support. Software applications are just efficient tools that should be seamlessly integrated with activities through ubiquitous user interfaces.

In order to satisfy these new requirements we need a paradigm shift from a system-centric to a user-centric view. What this means in particular is that we will have to abandon the desktop metaphor of personal computing. This metaphor, which has been highly successful, is built entirely around the notion of a virtual world, where the user context is defined by the state of applications and data. In a user-centric view, we recognize that users are highly mobile, interact with each other in physical environments, and interact with numerous devices, some or which are carried and some of which are ubiquitously available in the environment. We believe that the main impact of the user-centric approach will be that the boundaries of the systems we consider will become blurred, or at least much less static than today. Indeed, every electronic device the user may be in contact with will be potentially part of the system.

In addition to the user-centric view discussed above, ubiquitous applications exhibit another important characteristic, that of combining user-created data and user-initiated interaction with the capture of data from the environment by means of sensors. Application platforms therefore must deal with a range of devices and protocols that include both higher level services like Web services or databases, and device-centric services that need to run with very small foot-print. Clearly, a sensor that provides the tyre pressure runs with software that has a smaller footprint and a simpler interface than a Web service that provides information about the traffic situation on different motorways. Nevertheless, both may play an equally important role in a traffic security application, and should be accessible to the application developers through similar abstractions.

In this paper we present STITCH, a middleware for ubiquitous applications that addresses the concerns above. It relies on an integrated view of two mechanisms, namely distributed events and transactional resource manipulation. The former is well-suited to handle loosely coupled small devices, while the latter provides a basis for verification of distributed conditions and the triggering of coordinated actions. The STITCH infrastructure provides a layer of abstraction allowing software developers to take advantage of both paradigms.

Section 2 briefly describes the respective advantages of distributed events and transactional resource manipulation and how they may be combined. Section 3 highlights through a small example the flexibility of our middleware. Finally, Section 4 concludes the paper.

2. The Middleware

In this section we describe the two fundamental paradigms of our middleware: resource-based programming and event-based programming. Then we explain how they can be combined to cover different needs for interoperability among distributed components.

2.1. Resource-based Programming

Linda [1, 2], which first coined the word “coordination”, defines a paradigm for loosely coupled systems, built from the notion of tuple space. A tuple is an ordered set of values (e.g. an entity with a set of value attributes) and the tuple space is a storage space for tuples that is shared among a number of software services that communicate among themselves solely through manipulation of tuples. The notion of tuple space is nowadays a quite common infrastructural tool [3, 4], and its use for ubiquitous computing is appealing. When a location sensor detects

that a user is now in his car, this information can be inserted into the tuple space, only to be removed when a new location has been sensed. An application that need to know whether the user is in his car, would attempt to retrieve this information from the tuple space.

One of the major difficulties of programming paradigms based on shared tuple spaces is its need to be omnipresent and available to all services and applications. While omnipresence is feasible in a homogeneous environment, it becomes extremely difficult to implement when very diverse environments are involved in a coordination. STITCH borrows its coordination model with a distributed, omnipresent tuple space from our work on the CLF (Coordination Language Facility) middleware and its Mekano library [5, 6] , and leverages on this paradigm to provide developers with high-level abstractions that help them build distributed applications.

In CLF the global tuple space is distributed over all components, and the manipulation of tuples obeys strict rules. We usually refer to tuples in CLF as resources. In the CLF model, every component is a resource manager, which exposes a set of services that give access to its resources to the outside world. In fact, a CLF component may be seen as a tuple space where specific semantics are associated to the operations for insertion and removal of resources.

Resource manipulations are invoked through these services in accordance with the CLF protocol, which governs how services may be called. While such manipulations could in principle be coded "by hand", the preferred way to do this is to rely on the CLF scripting language. CLF scripts, which are based on production rules, are enacted by dedicated components called *coordinators*. The resource manipulations based on the CLF protocol is grouped into into three phases:

- *Search*: The search phase involves pattern-matching over the resources. During this phase, the system tries to find resources satisfying some properties by interrogating services of various components. For each matching resource an offer for action on this resource is returned. There may possibly be side effects within the component when this happens, but the nature of these side effects depend on the semantics associated by the particular component. A typical coordination rule will specify patterns involving resources managed by several components. A solution to a search is therefore represented as a set of combinations of returned offers from various components.
- *Performance*: As soon as complete solutions come out of the search phase, the coordinator tries to transactionally perform the actions associated with each of the matching resources. This is done by unrolling a two-phase commit protocol where the action is effectively performed at commit time. During the first part of the two-phase commit, components may deny previous offers (the service is no longer available, the user has left his car, ...), resulting in a transactional abort of all initiated actions. If no offers are denied, the transaction commits, in which case the resources are removed (they have been "consumed"). When this happens it is likely that there are side effects in the components from which resources have been consumed, but again the nature of these side effects depends on the semantics associated to resource consumption in each component.
- *Insertion*: When a solution has gone successfully through the performance phase, the transactional part of the resource manipulation operation terminates. This is followed by the insertion phase, where new resources are inserted into the resource space (the system ensures that resources will eventually be inserted).

A CLF rule consists of a left-hand side and a right-hand side. The left-hand side specifies the conditional part of the rule and translates into the two first phases of the resource manipulation, whereas the right-hand side specifies resource insertion and translates into the third phase. For instance, consider the following rule:

```
inCar('bob') @ msgAvailable('bob', msg)
<>- hifi(msg) @ inCar('bob')
```

In our rule language the "`<>-`" sign should be understood as implication and separates the left-hand from the right-hand side. The "@" sign can be thought of as "AND". The left-hand side is therefore a conjunction of primitive conditionals (on resources), referred to as "tokens". Tokens on the right-hand side designate resources to be inserted into the tuple space. In the rule above, the token `inCar` encapsulates the presence of a person in a car and `inCar('bob')` is true when bob is in the car. The token `msgAvailable` encapsulates a message delivery system and the token `hifi` encapsulates the hi-fi system of the car. Here is how the three phases would work:

- *Search*: The rule is triggered when the person inside the car is *'bob'* and there is a message for him.
- *Performance*: If it is possible to reserve at the same time the resources for the sensed user and the message then both will be atomically consumed.
- *Insertion*: A message resource will be inserted into the hi-fi encapsulator, which could play it through the car speakers. The resource *'bob'*, which has been removed from the tuple space, is also re-inserted in order to allow the rule to be triggered again (otherwise only a single message would be processed).

As a CLF rule is always active, any new message that becomes available is immediately taken into account.

So how is the `inCar('b')` maintained to correctly reflect Bob's location? The answer to this will be developed below but in short, presence detectors are in charge of this. When Bob's presence is detected in the car, the corresponding resource is inserted into the tuple space. Also, for every new location detected for Bob, any resources that reflect another location are removed. Thus when Bob leaves the car, the resource *'bob'* disappears and any subsequent messages would not be processed until Bob is again detected in the car.

These few concepts have proven to be sufficient to express the logics of a range of applications around workflow, e-commerce, document notification and delivery and job scheduling [5]. In this in paper we illustrate how they can be used to implement ubiquitous applications.

2.2. Event-Based Programming

Loose coupling between components can be achieved through different means. In the resource-based paradigm this is done by limiting the interaction between components to the manipulation of shared resources. Event-based programming is another paradigm for connecting components in a loosely coupled manner, where interactions take place through events sent and

received. The event-based model is widely used both in centralized and distributed settings. Basically, all components are connected to an event bus through which they can perform two basic operations:

- *Subscribe*: declare interest in some type of events. The subscription is maintained by the event bus, which afterwards will forward all relevant events to the subscriber.
- *Publish*: send an event to the event bus, which is in charge of forwarding it to all the interested subscribers.

It is important to understand that publishers and subscribers may not be aware of each other and that they do not even need to be simultaneously on-line to communicate, since events can be buffered in a queue by the event bus. This paradigm is easy to grasp: each component takes the responsibility of processing some type of events and to publish the results.

In STITCH, we have enriched our middleware with dedicated components which act as event buses, and a lightweight client library providing the publish and subscribe operations. This brings a simple and efficient way to asynchronously connect distributed components.

2.3. Combining Events and Resources

At first glance, resources and events may seem strikingly similar. However, one fundamental difference is what they represent: while resources represent state (the user is in his car), events represent state changes (the user entered his car, the user left his car). In the example above we could assume that presence detectors (e.g. RFID tag readers) will publish events when a user is detected. These events will be transformed into resources by components that on the one hand subscribe to events on an event bus and on the other hand expose a resource interface, which is visible to the coordination scripts of STITCH. Such components can be of varying complexity, and can for example combine received events from different types of sensors.

Another fundamental difference between events and resources comes from they way they can be manipulated and combined. Indeed, event-based programming is typically targeted towards systems where each component has the ability to satisfy a request on its own. As soon as we face the need to make several components work together towards a common goal, we have to lay additional multi-phase protocols on top of the simple publish-subscribe protocol. These issues of coordination are precisely the ones tackled by the resource-based programming paradigm. Hence, our approach of combining both paradigms in the same middleware, and to allow them to cooperate when implementing an application.

We have based our implementation of an event model on the existing resource-oriented capabilities of STITCH: events are represented as resources, event buses are resource managers, and the publish and subscribe operations are based on resource manipulations. There are two main advantages to this approach. First, it ensures a seamless integration between the two paradigms. Second, it allows us to manipulate events within the infrastructure; CLF rules can be set up to operate on events in order to combine them and to generate new types of events, all using the full power of resource manipulation, which includes transactional removal of sets of events in order to generate new events. Interestingly, events on different event buses can be combined this way, thus providing us with an important tool to scale up and to handle a large number of distributed event buses.

For instance, consider two sets of components, each one of them interoperating through a separate event bus, which provides a simple and fast asynchronous communication. Let's imagine that we want to be able to detect when two particular events are simultaneously published on the two event buses, and then act in consequence. We can do so by putting in place a coordinator with a single CLF rule. Its left hand side would search for the matching events on the two buses (recall that events are also resources and buses are resource managers), then transactionally consume them. The right hand-side of the rule would take the appropriate set of actions by inserting new resources. These actions may well include publishing an event in some event bus.

3. An Example

We consider here a very simple yet instructive problem: optimizing the use of elevators in office buildings. We focus on two possible optimization scenarios:

- First, as soon as the end of a meeting is detected, an elevator is sent to the floor where the meeting room is located. Hence, the elevator will be moving towards the floor while people will be walking towards the elevator. Latency is diminished by introducing parallelism between the two activities.
- Second, when several elevators are available, the number sent to the meeting floor will depend on the number of attendants and their probable destination (upstairs or downstairs). By analyzing more deeply the situation resources are better allocated.

3.1. General Architecture

Figure 1 describes the general architecture of the system. Two interoperability modes are clearly distinguished: event-based (dashed arrows) using a publish-subscribe protocol, and resource-based (plain arrows) using the CLF protocol.

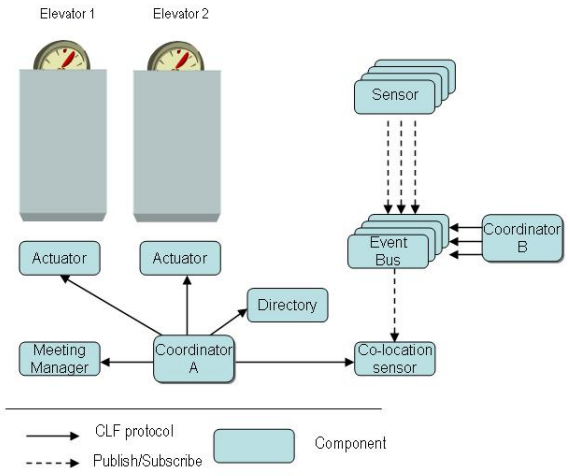

Figure 1: Architecture of elevator control system.

The sensors are connected to the event buses and publish basic events each time a user is detected in a room. The event buses and the location sensors are deployed across the building

in a hierarchical way. For instance, we could have one bus for the building, one for each aisle, or even one for each floor, depending on the number of sensors we want to attach to them. Each event bus also provides a resource interface which is used by the `Coordinator B` to coordinate events on the different buses. Through this mechanism, event buses are provided with filtering and forwarding capabilities that allow for smart propagation of events: at each level in the hierarchy events will be cascaded and/or escalated to the adjacent levels; certain events (for example a fire alarm) could be propagated to all buses.

The `co-location sensors` are *logical sensors* that aggregate basic location events in order to produce information about people being co-located. This information is in turn sent to an event bus but is also managed internally as resources that can be manipulated by the `Coordinator A` through a resource-based interface. In this example we are interested in managing the number of co-located persons at a particular location, as well as their identity. The component `Meeting Manager` is dedicated to this application and keeps track of how attendance evolves at each meeting.

3.2. Minimizing Latency

The start of a meeting can be detected by the presence of several people in a meeting room that may have previously booked. Relevant information is stored within the general `Directory` database, an enterprise system for managing information about users, calendars, meeting rooms, etc. Other information is deduced from the `co-location sensor`.

The end of a meeting can be assumed when at least half of the attendees have left the room. This information is obtained by comparing the maximum number of attendees with their current number (both available from the `Meeting Manager`), and helps us to avoid erroneously detecting the end of a meeting when only one person leaves the room and to miss the end if one or two people stay in the room after the meeting.

These two conditions can be adapted and expressed through CLF rules. When the conditions are realized the resource corresponding to the state of the meeting is set respectively to `'ongoing'` or `'finished'`. It is thus possible to have a rule triggered by presence of `'finished'` that inserts resources into the `actuators` associated to the elevators in order to send them to the right floor.

3.3. Optimizing Resource Allocation

In this scenario, we suppose that several elevators are available, that the meeting room is not located on the top or ground floors and that the "home" location of people (e.g. the location of their offices) is known by the `Directory` component. The list of people present can be obtained from the `Co-location sensor`. It is thus possible to predict the number of persons that is likely to move upstairs and downstairs. Depending on this and load indicators, one or more elevators will be sent.

3.4. Managing conflicts

So far we have assumed that only one meeting finishes at a given time. In reality it will be necessary to handle conflicting situations, where several meetings finishes at the same time, and will all be competing for controlling the elevators. Fortunately, our resource model, where resources are manipulated through a two-phase commit protocol, is suitable to handle such conflicts. By representing the functions provided by the elevator actuators as resources, we can assure that they are consistently managed. Thus, when a coordination rule has detected that all necessary resources are available (meeting is finished, elevators are available), it will transactionally consume the resources it needs, which may mean that another rule with conflicting needs will not see the resources and will therefore not be triggered.

4. Conclusion

By enhancing the resource-based programming capabilities of STITCH with an event model we have been able to offer a simple and lightweight interface for loosely coupled clients. This was a major requirement for quickly integrating new functions, devices and sensors. Moreover, events are very well adapted for dealing with mobility, disconnection and reconfiguration that are most common in ubiquitous environments. Other research efforts have also acknowledged the advantages of a multi-paradigm approach for building complex distributed systems, combining for instance agents and workflow [7].

As discussed, events are well-suited for connecting autonomous components, and can be implemented in a very light way. However, the event-based paradigm also has limitations and is not sufficient to achieve the functionalities we foresee in ubiquitous applications, and as we have shown in this paper, the resource-based paradigm is a highly useful complement to events. Indeed, the state of an entire system may be represented by resources that can be easily manipulated through the combination of coordinators and CLF scripts. With a couple of rules it is possible to describe how the system should react when the state changes. Here state is not limited to values that are stored in a central location but can be geographically distributed. This is particularly useful when it comes to capture and interpret context, and react accordingly.

Today we are in the process of designing and implementing several applications to test our approach in real settings. The current prototypes use a combination of real and simulated sensors and actuators, PDAs and desktop computers, allowing us to develop coordination scripts. By deploying hands-on actual ubiquitous environments we hope to gain further insights into which paths remain to be explored for the middleware.

5. References

[1] N. Carriero and D. Gelernter, "Linda in context," *Communications of the ACM*, vol. 32, no. 4, april 1989.

[2] N. Carriero and D. Gelernter, "Applications experience with Linda," in *Proceedings of the ACM SIGPLAN PPEALS, ACM Symposium on Parallel Programming*, New York, NY, 1988, vol. 23, pp. 173–187, ACM Press.

[3] J. Waldo et al., "JavaSpaces specification - 1.0," Tech. Rep., Sun Microsystems, Mar. 1998.

[4] P. Wyckoff, S.W. McLaughry, T.J. Lehman, and D.A. Ford., "T spaces," *IBM Systems Journal*, vol. 37, no. 3, april 1988.

[5] J-M. Andreoli, D. Arregui, F. Pacull, M. Riviere, J-Y. Vion-Dury, and J. Willamowski, "CLF/Mekano: a framework for building virtual-enterprise applications," in *Proc. of EDOC'99*, Manheim, Germany, 1999.

[6] J.-M. Andreoli, D. Arregui, F. Pacull, and J. Willamowski, "Resource-based scripting to stitch distributed components," in *Proc. of EDICS'2002*, Beijing, China, 2002.

[7] S.K. Shrivastava, L. Bellissard, D. Féliot, M. Herrmann, N. De Palma, and S.M. Wheater, "A workflow and agent based platform for service provisioning," Tech. Rep., European Commission, 2000.