Baton: A Service Management System for Coordinating Smart Things in Smart Spaces

Jingyu Li and Yuanchun Shi

Key Laboratory of Pervasive Computing, Ministry of Education,
Department of Computer Science and Technology,
Tsinghua University, Beijing 100084, China
lijingyu03@mails.tsinghua.edu.cn, shiyc@tsinghua.edu.cn

Abstract. Smart spaces are open complex computing systems, consisting of a large variety of cooperative smart things. Central to building smart spaces is the support for sophisticated coordination among diverse smart things collaborating to accomplish specified tasks. Multi-agent systems are often used as the software infrastructures to address the coordination in smart spaces. However, since agents in smart spaces are dynamic, resource-bounded and have complicated service dependencies, current approaches to coordination in multi-agent systems encounter new challenges when applied in smart spaces. In this paper, we present Baton, a service management system to explicitly resolve the particular issues stemming from smart spaces while coordinating agents (delegating smart things in smart spaces). Baton is designed as a complement to coordination approaches in multi-agent systems with a focus on mechanisms for service discovery, service composition, request arbitration and dependency maintenance. Baton has been now deployed in our own smart spaces to achieve better coordination for smart things.

1 Introduction

Smart Spaces [1] are typically open, distributed, heterogeneous and dynamic computing systems, which can be conceived as cooperating ensembles of a great variety of smart things, striving to achieve different missions. Accordingly, when building smart spaces, a fundamental aspect should be to support sophisticated coordination among diverse smart things working together towards accomplishing specified tasks. Many research projects have adopted multi-agent systems to enforce the coordination in smart spaces [2][3][4], where smart things are delegated by Agents, who provide services and consume services, and are coordinated in terms of service dependencies. However, existing coordination mechanisms of multi-agent systems need to be enhanced to cope with the particular situations when coordinating smart things in smart spaces [5].

1) Smart spaces are open and dynamic environments, where smart things (i.e. a smart phone) may enter or leave at will. Along with the smart things' appearance or absence, services provided by them are dynamically available or disappearing in a smart space, making the service consumers experience discontinuous services and thus tampering with the consumers' tasks. As a result, coordination in smart spaces

T. Enokido et al. (Eds.): EUC Workshops 2005, LNCS 3823, pp. 11 – 20, 2005.

needs to continuously maintain the service dependencies among smart things in spite of variations of service providers.

2) Smart things in smart spaces are resource-bounded since they are integrated with physical environments where physical resources are limited in number and have many physical constraints. So when there are more requests than a service provider can handle, for example, a video player, an email notifier and a file controller simultaneously require to use the only wall-sized display of a smart space, service request collisions will be incurred. Therefore, coordination in smart spaces should try to resolve request collisions and ensure that each consumer can get its deserved services so as to satisfy its service dependencies.
3) Besides the simple case in which a requested service can be directly provided by a single smart thing, smart spaces often encounter more complex situations where the requested service has to be fulfilled through the orchestration of multiple smart things conforming to certain control logic. A typical scenario is that a user may submit a PPT-Display service requirement, which must be satisfied by cooperating at least a File Reader and a Projector, or more considerately, a lamp controller (to dim the light for better vision) , a laser pointer (to control the PPT files) and so on. Consequently, to form new high-level services, coordination in smart spaces needs to comply with some coordination rules in addition to simple service dependencies.

Some of the existing multi-agent systems [13] perform agent coordination by means of high-level agent communication language and conversation protocols, such as FIPA ACL and KQML, which assumes that the interaction patterns are established in a priori and thus doesn't appropriately support the notion of openness and dynamicity [6]. OAA [2] provides a loose-coupling framework to accommodate dynamic agents and utilizes a "delegating computing" notion to coordinate agents, but doesn't deeply consider the problem of service request collisions. Metaglue [3] enforce its agent coordination with the assistance of a dedicated resource management system, called Rascal. Rascal [7] deals with many of the issues pertinent to smart spaces when coordinating agents, such as resolving request collisions, however, it doesn't take many considerations on composing several services to fulfill a potential request [12], which are common cases in smart spaces and doesn't pay much attention to maintaining service dependencies when agents join or leave smart spaces.

In this paper, we present Baton, a service management system to explicitly address the particular issues in smart spaces. Baton can be regarded as a complement to coordination approaches of multi-agent systems with a focus on mechanisms for service discovery, service composition, request arbitration and dependency maintenance. Baton is implemented on Smart Platform [4], which is a multi-agent system designed as the software infrastructure for our own smart spaces (Smart Classroom [8] and Smart Meeting Room [9]), where agents are loose-coupled and interact in a blackboard pattern.

The remainder of this paper is organized as follows: section 2 first defines some basic concepts, then presents an overview of the structure of Baton, and gives a detailed description of its two major components, *Knowledge Base* and *Service Broker*, which play key roles in addressing the particular issues when coordinating agents in smart spaces. Section 3 concludes the whole paper and discusses our future work.

2 Architecture of Baton

2.1 Basic Concepts

Before detailed introduction of Baton, we'd like to clarify several concepts relating to our work.

1. **Agents.** Agents are basic functional units in smart spaces, who provide services and in the meanwhile, consume services. Note that all the smart things are encapsulated as agents in our smart spaces. We assume that agents in smart spaces are trusted and friendly -- they can honestly express their service needs and capabilities, and release the services when they have finished their jobs or when some others need those services badly.
2. **Services.** Services are well defined functionalities provided by smart things, delegating by agents. An agent can provide one or more services, and a service may be provided by a number of agents. For example, a smart mirror may provide face recognition service and experience capture service, while a controlling service can be provided by a laser pointer or a speech recognizer. Services are the interfaces through which agents interact and cooperate with one another.
3. **Atomic Service** and **Composite Service.** An atomic service can be provided by a single agent, while a composite service has to be fulfilled through the teamwork of multiple agents. For example, in an automated office [14], a message notification service needs to be accomplished by at least a message receiving agent, a user location detection agent and a text or speech output agent.

2.2 Structure

The architecture of Baton is shown in Figure 1. Each of the smart spaces needs to run an instance of this architecture so as to perform its service management.

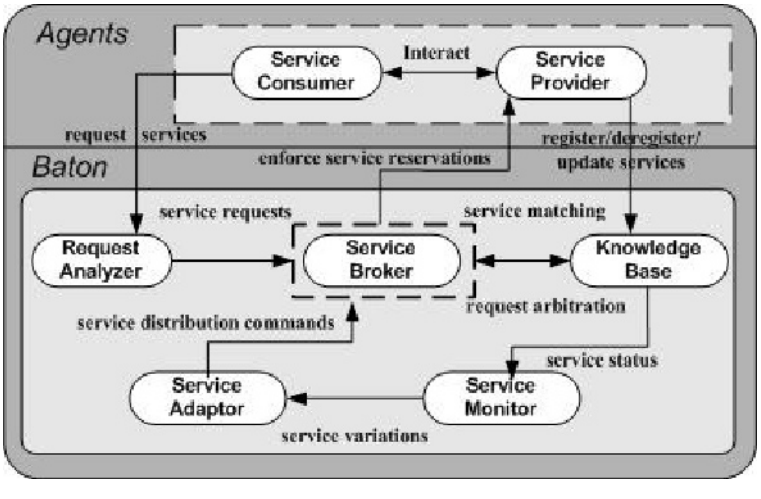

Fig. 1. Architecture of Baton

Baton consists of five components, which are *Knowledge Base*, *Service Broker*, *Request Analyzer*, *Service Monitor* and *Service Adaptor*. For the sake of efficiency, each of these components is encapsulated as a single agent, distributed among different devices and cooperating to fulfill the tasks of Baton.

Knowledge Base stores all the information about the services registered by agents, and performs the service-request matching. *Service Broker* is the core part of Baton, and takes charge of choosing the most suited services, deciding who should get the requested service when service request collision happens and constructing composite services. In fact, *Service Broker* (enveloped by dashed lines) may be extended to several distributed federated cooperating *Service Brokers* when the smart spaces become larger or there are more services. *Request Analyzer* translates the service requests into internal representations that can be parsed by *Service Broker*. *Service Adaptor* here is responsible for maintaining the service dependencies among agents in case of changes, and mainly considers two situations: (1) if a previous service request hasn't been satisfied, then once the desired service appears in the system, *Service Adaptor* will help to establish this service dependency; (2) if a service consumer loses its service while it is being served, *Service Adaptor* will try to find another substitute to continue this service. *Service Monitor* monitors and collects the service information in a real-time fashion. We are now extending *Service Monitor* from a component in Baton to a visualized tool, through which we can see clearly what kinds of services are active in a smart space, what they can do, and what status they are in.

In the following sections, we will give a further discussion on the two major components of Baton, the *Knowledge Base* and the *Service Broker*.

2.3 The Knowledge Base

Knowledge Base contains the information of both atomic services and composite services, and thus is the basis on which Baton makes all its decisions when coordinating agents. Atomic service information comes from the service descriptions submitted by agents when they first participate in a smart space, and is updated by agents themselves when change occurs in their lives. While composite service information is the knowledge about how to construct a composite service, which may include what atomic services are needed and what their logical relationship is to form this composite service.

Service Descriptions. We have recognized that the descriptions of services should mainly cover the following two aspects:

1) **Inherent information.** Inherent information describes the inherent features of a service, including service name, attributes and values, maximum capacity, provider, and service dependencies, which specify what other services are needed to provide this service, for example, the Speaker Recognition agent in Smart Classroom [8] often depends on the aid of the User Profile agent to correctly recognize the speaker. Inherent information also includes the interface information, which specifies how the service should be accessed and interacted with. With the purpose of achieving the automatic service invocation and interoperation, we utilize OWL-S [15] as the description language to describe the inherent information of a service.
2) **Dynamic information.** Dynamic information reflects the runtime states of a service, which describes to what extent the service has been used and how many of its

service dependencies have been satisfied. A service can be *free*, *reserved*, *busy* or *busy reserved*, its dependencies can be *satisfied*, *unsatisfied* or *satisfying*, and its current available capacity varies with its workload. Dynamic information is initialized as *free*, *unsatisfied* and *maximum capacity*, and is dynamically updated by the service provider in case of changes. Since OWL-S doesn't take many considerations on the runtime status of a service, we use XML language to describe the dynamic information to supplement the description of a service. Baton keeps track of the services by examining the descriptions of their dynamic information, which reveal every detail of the services in their whole lives.

Knowledge Representation for Composite Services. To accurately express the complicated relations among atomic services cooperating to perform different composite services, we borrow the idea of ConcurTaskTrees [10] to model a composite service. The tree-like structure with relational operators proposed by ConcurTaskTrees guarantees the integrity and clarity of the specifications of a wide variety of composite services. The nodes and relational operators used in the model of our composite service trees are defined as follows:

1) **CS.** CS delegates a composite service to be constructed, which can be a root node or an internal node of a composite service tree.
2) **AS.** AS is an atomic service, always being a leaf node of a composite service tree.
3) The operators describing the temporal relationships of the services are only applied to those on the same level of a composite service tree. In view of the current smart spaces, we only define four operators:

- **S1 >> S2:** Service S2 is activated when S1 terminates. For example, in Figure 2(a), as with a PPT-Display service, the projecting service only makes sense after the File-access service finishes.
- **S1 []>> S2:** When service S1 terminates, it provides some values for S2 besides activating it. A typical example is shown in Figure 2(b), a Bitmap-to-PPT transformation service can be performed by a Bitmap-to-*x* service first, and then an *x*-to-PPT service. Here *x* can be any transitional format, such as gif format.
- **S1 | S2:** choosing. As is shown in Figure 2(a), PPT-controlling can be performed by hand-free controlling like speech command, or manual controlling like pointing.
- **[S]:** S is optional. In Figure 2(a), marking service is optional and is activated only if the consumer needs to make annotations on the PPT file.

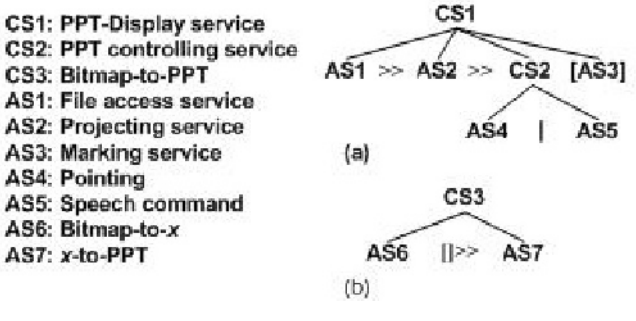

Fig. 2. Examples of composite service trees

The knowledge for a composite service may either be well-predefined by application designers, who just need to specify the composition strategy by using the ConcurTaskTrees model (e.g. Fig.2.(a)), or be generated based on the OWL ontology of the atomic services (e.g. Fig.2.(b)).

In terms of our own smart spaces [8] [9], a number of possible composite services have been modeled using the above nodes and operators. And for consistency, the ConcurTaskTrees models are also mapped to OWL-S descriptions. For convenience and efficiency, the Berkeley DB XML [11], a dedicated database for mastering xml files is introduced in Baton as the *Knowledge Base* to maintain the descriptions of all services, and a query language, Xpath [16] is utilized and extended to access the database.

2.4 The Service Broker

Evaluation for Service Matching. In spite that there may be several candidate services for a request found in *Knowledge Base*, it is probable that some of them may be insufficient and some may be an oversupply. For example, as to a request for a color printing service, if a color printer doesn't exist, a black-and-white one can be a substitute but is inadequate; while as to a request for a video playing service with maximum frame rate of 30fps, the service offering 40fps will be a waste. Actually, there exists an affinity between a service and a request --- the more perfectly the service matches the request, the closer the affinity. Given a request R, a candidate service S, then $Affinity(R, S) = Match(R, S)$, where the value of $Match(R,S)$ is estimated according to several predefined matching rules, $Match(R, S) \in [0, 1]$.

In simple cases, *Service Broker* just picks the free service owning the highest *Affinity* value above a preset threshold as the final choice for a request. However, if all the candidate services are being occupied, the process of solving request collisions will be activated.

Solution for Request Collisions. Services provided by agents in smart spaces are capacity-limited, so if multiple service consumers request the same service, a verdict must be made on who should get the service. While solving request collisions, we recognize that three guidelines should be followed in smart spaces:

1) Agents have different priorities in smart spaces, for example, an agent delegates a teacher in Smart Classroom has a much higher priority than those delegating students. So when different agents contend for the same service, priority is a very important factor in deciding to whom the service should go.
2) Agents in smart spaces tend to be served continuously, rather than frequently disturbed. So if request collision occurs and service redistribution is inevitable, then the changes of service dependencies brought by the redistribution should be minimized.
3) Service types in smart spaces are varied, so it is difficult to achieve a global optimal distribution of all services, but reaching an optimal distribution of a single service is quite feasible.

According to these guidelines, we take subsequent considerations:

– Service consumers usually have contentment evaluations on the results of their service requests. Specifically, a contentment evaluation basically relies on whether the consumer can get its desired service, and how the affinity between the request

and the service is like. A formal description of the contentment evaluation is: Given a request *R*, a service *S*, then

$$Contentment\ (R, S) = Affinity\ (R,\ S) \times Available\ (R,\ S) \tag{1}$$

Available(*R, S*) demonstrates that whether service *S* can be used by *R*, if yes, then *Available*(*R, S*) = 1, otherwise, *Available*(*R, S*) = 0.

– However, since S*ervice Broker* has to dynamically adjust the distributions of services in response to requests submitted by agents at will, consumers' contentment evaluations on a certain service vary from time to time. To be specific, for example, when the *Nth* request on service *S* comes, consumer A's contentment evaluation on *S* may be 0, as its request for *S* hasn't been fulfilled; whereas, when the *N*+1*th* request on *S* comes, A's contentment evaluation may increase to 1 because it has acquired *S* for some reason, and when the *N*+2*th* request comes, *S* may be taken away from A, making A quite displeased, and A's contentment evaluation may decrease to -1. To reflect that contentment evaluation is changing with new coming request, we formulate contentment evaluation with a variation of equation (1):

$$Contentment\ (R, S, N) = Affinity\ (R,\ S) \times f\ (R, S, N) \tag{2}$$

N indicates the *Nth* request on S and $f\ (R, S, N)$ is defined as follows:

$$f(R,S,N) = \begin{cases} 1 & if\ \ Available(R,S,N) = 1 \\ 0 & if\ \ Available(R,S,N-1) = Available(R,S,N) = 0 \\ -1 & if\ \ Available(R,S,N-1) = 1 \,\&\, Available(R,S,N) = 0 \end{cases} \tag{3}$$

– The goal of *Service Broker* in solving request collisions is to take every effort to fulfill each request for a certain service so as to maximize the total contentment evaluations of all the consumers on this service, and ensure that this service is not exploited beyond its capacity. Therefore, the problem of solving request collisions turns out to be a constraint satisfaction problem. It is reasonable to believe that when two consumers request the same service, distributing the service to the one who has a higher priority will take more benefits to the sum of contentment evaluations, so the priority value of each consumer can be assigned as the weight of its contentment evaluation when calculating the totals. Consequently, as with a given service *S*, when the *Nth* request on *S* comes and incurs a collision, the goal of *Service Broker* can be clearly illustrated as:

$$\begin{aligned} &Max\ C\ (S, N) = \Sigma\ [Priority\ (Ri) \times Contentment\ (Ri, S, N)\ ] \\ &\text{s.t.}\ \ \Sigma\ [Require(Ri, S) \times Available\ (Ri, S, N)] \le Maximum\ Capacity(S); \\ &\quad Ri \in RA = \{Requests\ on\ S\ from\ service\ consumers\}; \\ &\quad Available\ (Ri, S, N) \in \{0,1\}; \\ &\quad i = 1, 2\ \ldots .|RA|; \end{aligned} \tag{4}$$

Therein, *Require*(*Ri, S*) specifies the requirement that *Ri* poses on *S*, which will be discussed later.

– In terms of capacity, we identify two distinct categories of services:

1. Capacity of a service means the largest number of consumers that the service can support in parallel. For example, the capacity of a speech recognition agent may be that it can simultaneously handle three channels of speech stream. As to this case,

any consumer can only get a copy of the service, thus *Require*(*Ri*, *S*)=1, and the constraint condition in (4) can be simplified to:

$$\Sigma\ Available\ (Ri,\ S,\ N) \le Maximum\ Capacity(S) = 3.$$

2. Capacity of a service may have no explicit confinement on the number of consumers, but is limited by its own capability. For example, a video on-demand agent can provide video data accessing service with a bandwidth of at most *1000KBps*, and it can serve any number of consumers as long as the total sum of the bandwidth used by these consumers doesn't exceed *1000KBps*. In this case, *Service Broker* only checks whether the service can satisfy the minimum need of a request, and thus *Require*(*Ri*, *S*) equals the minimum requirement that *Ri* poses on *S*, for example, consumer A may request a video data accessing service with a rate at least *300KBps*, consumer B may request at least *400KBps* and consumer C at least *500KBps*. Thus the constraint condition in (2) will be expressed as:

- $300 \times Available(Ra,S,N) + 400 \times Available(Rb,S,N) + 500 \times Available(Rc,S,N) \le 1000.$

As a matter of fact, looking into equations (2), (3) and (4), we can see that only *Available(Ri, S, N)*s are variables, therefore, solving a request collision turns to be solving a simple linear programming problem with the variable domain to be {0,1}. The solutions can demonstrate which consumer can get the desired service when there is a collision. For example, a solution, *Available*(*Ra,S,N*)=0, *Available*(*Rb,S,N*)=1, *Available*(*Rc,S,N*)=1, means only consumer A can not get service S.

Algorithm for Service Brokering. A service request is handled by *Service Broker* as a transaction because we believe that the process of satisfying a request should be

```
function SatisfyRequest (rd) returns a service aggregation or failure;
    inputs: rd, request description from a consumer;
    return Commit(GetService(rd)); // two phase commit

function GetService(rd) returns a service aggregation;
    SA = {}; initial service aggregation
    service ← the service that matches rd;
   if service doesn't exist, then SA={}, return SA;
   if service is an atom service, then SA = SA U {service}, return SA;
   if service is a composite service, then
          {
            traverse the tree of this composite service using preorder traversal;
            for each node i in the tree do
                rdx ← service description of node i ;
                GetService(rdx);
          }

function Commit (SA) returns a solution or failure;
      inputs: SA, service aggregation containing all the desired atom services to
construct a composite service;
      if SA = {}, return failure;
      reserve all the services in SA;
      if any collision occurs when reserving, then solve the collisions using linear
programming model;
      if all services are reserved successfully, then return SA;
      if any service can not be reserved, then return failure;
```

Fig. 3. Algorithm for service brokering

indivisible, or we say atomic. *Service Broker* adopts a two-phase commit algorithm to guarantee the atomicity of the procedure of satisfying a request, in which *Service Broker* first collects and then reserves all the requested services, and according to the reservation responses from service providers, decides whether the request can be fulfilled. A short description of the algorithm is shown in Figure 3. For a composite service, when all its desired atomic services are available, *Service Broker* will take steps to coordinate agents providing these services to perform the composite service based on its knowledge description.

3 Conclusion and Future Work

When multi-agent systems are situated in smart spaces to address the coordination of various smart things, agent coordination approaches encounter new challenges. In this paper, we present Baton, a service management system to enhance the coordination mechanisms of multi-agent systems in smart spaces. Services in Baton are described by OWL-S language, which makes the processes of service discovery and composition more accurate and efficient. Solutions for request collisions are modeled as simple linear programming problems, which makes it easy to solve the collisions and in the meanwhile, keep changes of service dependencies to the minimum. The process of fulfilling a request is handled as a transaction, and a two-phase commit algorithm is utilized to assure its atomicity. Currently, Baton has been built into our Smart Classroom [8] and Smart Meeting Room [9] to manage the services of the systems so as to sustain better coordination of the smart things in smart spaces.

We are now trying to improve the dynamic service composition strategy by using the semantic information of services, and will add proper access controls of services to Baton so as to settle the security problem in smart spaces.

References

1. NIST Smart Space Laboratory. http://www.nist.gov/smartspace
2. David L. Martin, Adam J. Cheyer, Douglas B. Moran: The open agent architecture: A framework for building distributed software systems. Applied Artificial Intelligence, 13(1-2): 91–128, January-March 1999
3. Brenton Phillips. Metaglue: A programming language for multi-agent systems. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, 1999
4. Xie W K, Shi Y C, Xu G Y, et al: Smart Platform - A Software Infrastructure for Smart Space (SISS). The Fourth International Conference on Multimodal Interfaces, Pittsburgh, USA, 2002
5. Andrea Omicini, Sascha Ossowski: Objective versus Subjective Coordination in the Engineering of Agent Systems. The AgentLink Perspective: pp. 179 - 202
6. Martin Fredriksson, Rune Gustavsson, Alessandro Ricci: Sustainable Coordination. The Agent Link Perspective: pp. 203 - 233
7. Krzysztof Gajos: Rascal – A Resource Manager for Multi-Agent Systems in Smart Spaces. CEEMAS01, Krakow, Poland, 2001
8. Yuanchun Shi, Weikai Xie, Guangyou Xu, et al: The Smart Classroom: Merging Technologies for Seamless Tele-Education. IEEE Pervasive Computing, vol. 2, no. 2, pp. 47-55, 2003

9. Xin Xiao, Enyi Chen, Yuanchun Shi: Multimedia Communication between Mobile Devices and Smart Spaces. The 13th National Multimedia Conference, Ningbo, China
10. F. Paterno, C. Mancini, S. Meniconi: Concur-TaskTrees: A Diagrammatic Notation for Specifying Task Models. Proc. Interact, Sydney, 1997
11. Berkeley DB XML Download page. http://www.sleepycat.com/download/xml/index.shtml
12. Robert Kochman: Decision Theoretic Resource Management for Intelligent Environments. http://www.csail.mit.edu/research/abstracts/abstracts03/interfaces-applications/interfaces-applications.html
13. Jade Technical Overview. http://www.jadeworld.com/downloads/Jade6/technicaloverview
14. Automated Office. http://www.ai.sri.com/~oaa
15. OWL-S 1.1 Release. http://www.daml.org/services/owl-s/1.1
16. XML Path Language Version 1.0. http://www.w3.org/TR/xpath