

Smart Platform - A Software Infrastructure for Smart Space (SISS)

Weikai Xie, Yuanchun Shi, Guanyou Xu, Yanhua Mao

Department of CS, Tsinghua University, 100084, Beijing, China
{xwk@media.cs, shiyc@, xgy-dcs@, myh@media.cs}.tsinghua.edu.cn

Abstract. Building a Smart Space still remains a low-efficiency task. Among others, the lack of an adequate Software Infrastructure for Smart Space (SISS) accounts for this situation in a great part. Previous proposed solutions for SISS did not sufficiently address the issue of performance and usability. A new solution, Smart Platform, which is focused on improving these aspects of a SISS, is presented in this paper. To optimize the performance of the inter-module communication, the stream-oriented communication is distinguished from the message-oriented ones, and a corresponding hybrid communication scheme is proposed. To improve the usability, a featured loose-coupling structure, a straightforward Publish-and-Subscribe coordination model as well as a set of user-friendly deployment and development tools are developed. Besides, Smart Platform is intended as an open and generic SISS available for other research groups. To this end, XML-based message syntax and open wire-protocol based architecture are adopted to make sharing research efforts more easily. The effectiveness and efficiency of Smart Platform has been validated by its practical use in the Smart Classroom project and some evaluation experiments. The interoperation between heterogeneous SISSs will be a demanding need as more and more Smart Spaces are built. A dual-citizenship agent approach to address this issue is proposed in the last part of the paper.

1 Introduction

A Smart Space system will typically involve dozens of distributed modules that usually are not dedicated for running together, such as speech recognition, human-tracking and gesture recognition. As stated by many researchers [1][2], there is a clear need of a software infrastructure to enable these distributed modules to connect, communicate with each other and further to collaborate with each other in a coherent and structured way. It has been recognized by the researchers in this field that those generic-purpose distributed computing architecture is not suitable here. For example, Brumitt et al. described the deficiency of DCOM, Java and CORBA in this sense [2].

1.1 Related works

Recently there have been some research efforts devoted to develop a software infrastructure dedicated for Smart Space (For simplicity, we will refer to Software Infrastructure for Smart Space as SISS in the following). The most well-known ones are, OAA from SRI [3], Metagluue from MIT [1], Event Heap from Stanford [4] and InConcert from Microsoft Research [2].

OAA (Open Agent Architecture) is a multi-agent system framework for integrating a community of heterogeneous software modules in a distributed environment. Its main notion is so called “delegating computing”, which means all the messages between agents are mediated by a central component called Facilitator. The message format and agent-coordination model is modeled after Prolog language and distributed problem solving.

Metagluue is an extension to the Java programming language for building software agent systems to control Smart Spaces. Agents communicate with either RMI of Java or a self-made event broadcast mechanism. A capability-based approach is adopted to address agents. An agent can be dynamically loaded and unloaded without interrupting other agents who have established connections with it. An agent can also store its state and retrieve back its state in a persistent storage.

Event Heap provides a low-level communication and coordination mechanism based on Tuple-Space model. The message format is a tuple, which is essentially a set of ordered typed fields. Modules communicate by posting tuples to a central shared storage and retrieving them from the storage if they match some pattern.

InConcert is a middleware platform, which provides asynchronous message passing, machine independent addressing and XML-based message protocols to reduce the effort required to build individual components that can communicate in the distributed environment

1.2 Our effort

About two years ago, we started the project Smart Classroom [5] to build a smart space for Tele-education. In its first stage, we adopted the OAA as its supporting SISS. But we found it could not satisfy our needs well, especially in its performance. In a Smart Space it is a common need that modules should exchange messages in real-time and frequently. For example, a human-tracking module might need to broadcast the user’s location every 100 ms. However, the average delivery latency of a message in OAA is above several hundred seconds and the system blocks up easily if an agent sends messages faster than ten messages per second or so. As far as we know, other proposed SISS did not give adequate considerations to the issue of system performance either. Besides, we found most of these systems are not user-friendly enough. To deploy them and grasp their usage usually requires a lot of efforts, bringing a serious problem in the development of Smart Spaces since most module developers there are not experienced with distributed system.

In this light we developed a new SISS, Smart Platform, with focus on improving these two aspects, while to make it a generic frame. Smart Platform has demonstrated

its advantages compared with other presented SISS solutions. Currently our Smart Classroom has been successfully ported to and running steadily on the Smart Platform. As one of our original motivations, we have also released a beta version of Smart Platform for downloading from our website [6].

The other contributions of our work includes: 1) We systematically analyzed the role of SISS and the requirements it has to meet, and thus build a common framework for evaluating the works in this field. 2) We suggested to distinguish two kinds of requirement on communication service in Smart Spaces: message-oriented and stream-oriented, and put forward a hybrid scheme to provide optimized support to both kinds of communication. 3) We studied the issues of the interoperation between heterogeneous SISSs, such as Smart Platform and Metaglug.

In the rest of this paper, we first describe our considerations on what SISS is and what its necessary features are. Then we present the Smart Platform in detail, including its structure, communication model, message format, services and ADK, as well as the comparisons with other related works (except InConcert, for the lack of its detailed document). To testify the effectiveness and efficiency of Smart Platform, we illustrate its practical application in the Smart Classroom project, and present a performance comparison conducted on it. Finally, we introduce our preliminary research results on the interoperation between heterogeneous SISSs.

2 The Software Infrastructure for Smart Space

2.1 Role of SISS

The architecture of Smart Space is composed of, from bottom up, underlying OS/Network on each participating computers, the SISS, and the distributed functional modules running on them. The role of SISS in a Smart Space could be considered from two perspectives, as illustrated in Fig. 1.

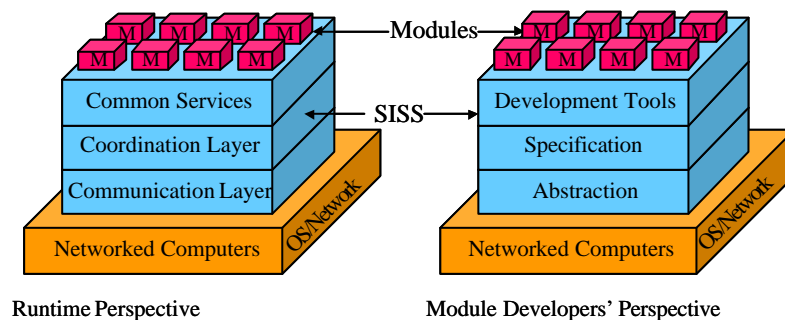


Fig. 1. The role of SISS from two different perspectives

First, from the runtime perspective, SISS serves as a runtime environment that is layered on the top of OS and network. Its internal structure can also be divided into three layers. The lowest layer is the **communication layer**, which provides the communication capability for distributed modules, supporting them to connect and communicate with each other. In this layer, the Quality of Service (QoS) of the communication is the most concerned. The middle layer is the **coordination layer**, which enables all the distributed modules in a Smart Space to collaborate in a structured and coherent manner for achieving the overall goal of the system. In this layer, the coordination model, which defines how the individual interactions between agents are organized in microscope, is the most concerned. In the upper layer, there are some **common services** that are shared system-wide. Those services required by most Smart Space systems are considered as part of SISS, including, among others, directory service for locating modules, configuration data service for maintaining the configuration data of the distributed modules in a centralized way, user profile service for management of the user preferences.

Second, from the module developers' perspective, SISS provides **abstraction, specification and development tools** for them to build modules that could be assembled in the Smart Space. Smart Spaces usually involve works from many different areas of computer science, which are developed in different programming languages and with different forms (say, source code, binary library, executive code). For the separation of concerns, developers need some well-designed **abstractions**. **Specification** defines the convention and protocol that a module should comply with in order to be integrated into the whole system in runtime. **Development tools** reduce the effort required to build modules that comply with the **specification**. It provides reusable codes that encapsulate the low-level implementation according to **specification** in some high-level forms, such as base class, static library or DLLs. Development tools also includes the built-in mechanisms in SISS and some common services that ease the debugging work, such as tracing out the communication details between any specified modules.

2.2 Some necessary features of SISS

Coen, et al. has given a comprehensive description on the characteristics of Smart Spaces [1]. We extended his work by identifying the specific requirements on a SISS introduced by these characteristics, which distinguish them from other distributed computing systems.

Loose coupling. A Smart Space system is very dynamic in itself. Modules are restarted or moved to different hosts usually. System configurations are changing time to time. The case is even worse during the developing period. If every single change occurred in the system requires every other modules in the system to be restarted, it would be a nightmare. The key to accommodate this nature of Smart Space here is to introduce the loose coupling feature into SISS, which is also beneficial for the resilience to failure.

Extensible. Being extensible, means that a SISS should allow the systems built on it to extend and expand gradually, for the cumulative building strategy is often adopted in

the development of a Smart Space. SISS should support introducing new modules on the fly and refining message definitions without impacting prior modules.

Support for stream-oriented communication. Coen, et al. has pointed out the communication in Smart Space should ensure real-time delivery [1]. Here we give a further consideration on the QoS that the communication layer of SISS should provide.

We argue that actually there are two catalogs of communications needs in Smart Spaces that have different requirements on the QoS of communications.

First catalog, named as **message-oriented communication**, is those communications that occasionally happens and usually have high-level semantics, e. g. to issue a command asking a module to turn on the light that it controls, or asking a speech-recognition module to inform others when a predefined keyword is recognized. These communications are sensitive to the loss of messages; whereas their requirements on the delivery latency is moderate, as long as it is within a reasonable boundary, say 50 ms, according to the human's cognitive character.

Second catalog, named as **stream-oriented communication**, is those communications that constantly occurs. Their semantic level usually is relatively low and the drop of data units up to several is usually tolerable. But they are sensitive to the variation of the delivery latency. However in most cases their requirement on the delivery latency itself is also moderate. An example is the delivery of captured video to multiple computer vision modules across network. Another example can be found in our Smart Classroom, where users can drive the pointer on a wall-sized display by pointing to the screen and then moving his hand (We call this facility Virtual Mouse). The communication between the module that track the hand movement and the module that actually control the pointer on the display just falls into this catalog.

Previous works on SISS usually only address the first catalog of communication needs. However it is equally important to support the second catalog of communication needs in SISS. More specifically, the variation in delivery latency could be guaranteed (this feature is also called isochronous) if asked.

Support for one-to-many communication. It is a common requirement in Smart Spaces that a message should be delivered to many modules simultaneously. Even in the case of stream-type communication, there will be multiple consumers of a specific stream data, such as the video-dispatching example mentioned above. Therefore it is more convenient for the module developer to build the one-to-many into SISS as its integral capability. One-to-many communication does not have to be implemented in network layer multicast.

Support for heterogeneous platforms and implementation languages. The modules in a Smart Space usually impose different requirements on the underlying hardware and software platform. Moreover they are often implemented in different language. A SISS should have the adequate capability to deal with all these diversities.

Lightweight. Last but not least, a SISS should be lightweight. As mentioned above, the development of a Smart Space involves many researchers from different areas. These researchers may not have enough backgrounds on the distributed computing as the designers of SISS themselves, and most of them are not willing to spend too

much time on learning how to deploy the SISS in their working environments and how to use it. Therefore whether the structure and the development interface of a SISS are straightforward enough is a practical factor for its success.

3 System Structure of Smart Platform

3.1 Abstraction Model

Distributed Component Model (DCM) and Multi-Agent System (MAS) are two candidates of abstraction models for constructing a distributed system. In DCM model, the basic encapsulation structure, object (distributed component is essentially distributed object), is passive, i.e. it should be activated or invoked by other process, which implies a centralized thread of application logic should exist in the system to invoke these passive objects when necessary. Architectures that follow this model include DCOM, CORBA and EJB. However, this model has one main shortcoming in the case of Smart Space in that it is hard to find such a single clear thread of application logic in a Smart Space. Instead, there are always many parallel threads of application logic to occur. For example, a human-tracking module should continuously track the position of the person, while a speech-recognition module should keep recognizing the user's utterance. In contrast, in the MAS model, each constituent module, agent, has its own execution process and the overall system functions are carried out by the cooperation among them, which matches the nature of Smart Space better. Therefore we adopted MAS model as the abstraction model for Smart Platform.

In Smart Platform, agent is the basic constitutive unit, which could be any software modules, for example, a device driver, a complicated perception algorithm or high-level application logic, as long as it can run autonomously and comply with the specification defined for an agent. Agents communicate with each other with the support of the communication layer of Smart Platform. The intention of a communication effort might be notifying others a happened event, querying others for a value, replying to the query, or anything else appropriate.

OAA and Metaglu are also based on the MAS model. According to our understanding, Event Heap is essentially based on the MAS model too, although its developers do not clearly state that. In addition, the agent in Metaglu has some limited capability of mobility. This is a major advance of Metaglu compared with others.

3.2 Runtime Environment Structure

Fig. 2 illustrates the overall runtime structure of Smart Platform. Besides normal agents, there are another two special system-level components provided by Smart Platform – DS and Container.

1) **DS** (the name initially came from the abbreviation of Directory Service). A Smart Platform runtime environment should run one instance of this component globally. It

lies in the center of Smart Platform and provides several key functions in the system, such as the registration of the agents, directory service, message dispatching. For a lightweight system structure, several common services are also integrated into the DS program. The functions of DS will be explained in detail in later sections.

2) **Container.** Each computer that participate the Smart Platform should run an instance of Container as a daemon process. An agent actually only contact with the Container on its host when it needs to interact with other parts of the system and the Container will contact with the DS or other related Containers on behalf of the agent to achieve its goal. Put it another way, Container builds a local environment for the agents running on the same host, and make the rest of the world transparent to them. This level of indirectness brings two main benefits. First, it restricts the distribution of prior knowledge of the system configuration (such as where the DS is running) to a single point (the Container) per host, which caters for loose coupling. Second, it provides a central point of administration for all the agents on a host. For example, Container provides interfaces for users to launch or terminate agents on its host.

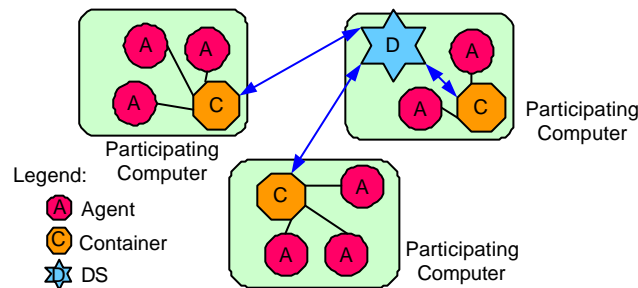


Fig. 2. Runtime Structure of Smart Platform

As far as we know, OAA and Event Heap do not have mechanisms that can provide the same functions as the Container in Smart Platform does, so they do not have the advantages that Smart Platform possess by the introducing of Container. The Metaglug Manager agent together with its associated MVM (Metaglug Virtual Machine) in Metaglug plays an equivalent role as Container in Smart Platform.

3.3 Communication Layer

Hybrid Inter-agent communication scheme

The inter-agent communication in a MAS system usually is carried out in two different schemes. One scheme is so called **direct scheme**, where each pair of agents that need to communicate should establish a direct connection. Another scheme is so called **mediated scheme**, where each agent only need to establish a connection with a centered system-level component, "broker", and all the communications are mediated by this "broker" component.

The mediated scheme has several advantages compared with the other one. First, it decouples the source party of the communication from the destination party, and thus caters for the loose-coupling nature expected. Second, it is more natural to accomplish the one-to-many communication. Third, the number of connections that should be maintained is linear with the number of the agents, much less than that in the direct scheme. However since all the communication flows should be processed and dispatched by the broker component, the delivery latency, especially its variation, of any single connection is hard to guarantee. Or put it another way, this scheme cannot provide the required QoS for stream-oriented communication.

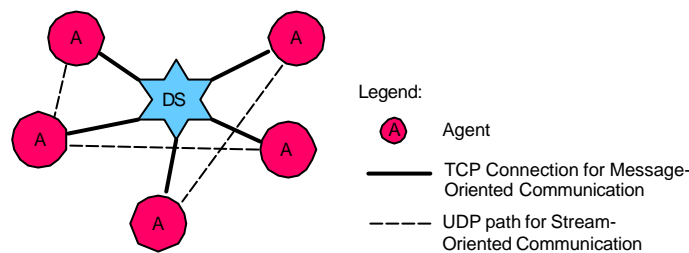


Fig. 3. Hybrid communication scheme in Smart Platform

Therefore Smart Platform adopts a hybrid scheme in its communication layer so as to provide an optimized communication service for each catalog of communication, namely all agents keep a connection with the DS (Actually there is a Container lies between the connection, but this fact can be neglected here), and at the same time is ready to accept direct communications by listening on some UDP ports, as illustrated in Fig. 3. For message-oriented communication, messages will be sent to the DS and forwarded by it to the destination. Since the connection is based on TCP, the messages can be reliably transferred. For stream-oriented communication, the messages will be directly sent to the destination agent via UDP. Actually, a multicast address is used to achieve the one-to-many communication here. Since no other intermediate hop is involved, the delivery latency is minimized. Furthermore, we are currently working on adopting RTP (Real-time Transport Protocol) [7] and some associated smoothing algorithm instead of the plain UDP used now to better guarantee a bound variance in the delivery latency that is critical to stream-oriented communication.

It is obvious that the hybrid scheme developed in Smart Platform will be superior over either mediated or direct scheme alone as used in OAA, Metaglu and Event Heap (More specifically, OAA and Event Heap adopts the mediated scheme; Metaglu adopts the direct scheme). Moreover, the support for stream-oriented communications in Smart Platform is unique in comparing with the other systems.

XML-Based Message Format

In Smart Platform, no matter for message-oriented communication or stream-oriented communication, a message is the basic data unit of the communication. Messages in Smart Platform are expressed in XML. If the original data is in binary format, such as a

frame of video, Base64 encoding is used to convert it to ASCII format. The adoption of XML has some significant advantages, which has been convinced by the real use of Smart Platform in our Smart Classroom project.

1) The good extensibility of XML contributes greatly to the extensibility of Smart Platform. Developers can start with a rough message structure and add detailed fields later without impacting prior works.

2) XML is user-friendly for its readable format and self-describing capability, i.e., describing the meaning of a field in addition to its value. This is very helpful when prototyping the system or debugging the system.

3) Developers can formally define valid messages in XML DTD, and we are building a XML validator into the debug facilities of the Smart Platform so as to find out the wrong-formatted messages that agents might send earlier. We have noticed that the mistyping of message by developers accounted for a great part of bugs found in a distributed system.

4) As an Internet standard, XML will facilitate to interoperate with other heterogeneous SISSs in future.

Of course, XML scheme is slightly inefficient compared with those binary-coding scheme, but we think it is trivial, given the advantages it brings and today's progress of computer power and network bandwidth.

Fig. 4 shows an example of messages used in Smart Platform, which may be generated by an agent in charge of speaker-recognition.

```
<SpeakerRcgResult>
  <Person Name="user_name1" Score="score1"/>
  <Person Name="user_name2" Score="score2"/>
  <Person Name="user_name3" Score="score3"/>
</SpekerRcgResult>
```

Fig. 4. A message in Smart Platform

OAA use a message format based on Prolog, which manifest several disadvantages. First, it has no adequate extensibility. Second, is relative difficult to grasp. Third, it is not self-describing. The message in Metaglu is represented as runtime object, which is not as user-friendly as XML is and is not self-describing as well. The message format in Event Heap is based on Tuple, which is essentially a set of ordered typed fields. The main shortcoming of this message format is it is not able to describe hierarchical data. At last, the message formats used in these systems are weak in supporting interoperation with heterogeneous systems.

3.4 Coordination Layer

Just as mentioned in above, coordination layer is at a higher and more global level than communication layer in a SISS. The latter cares how the communication between any two agents is carried out, whereas the former cares how the overall coordination of the system is achieved. Coordination model determines the style of communication services primitives of a SISS.

Publish-and-Subscribe Model

Generally speaking, all possible coordination models in MAS can be classified into four catalogs: Direct, Blackboard, Tuple-Space and Publish-and-Subscribe (also known as Meeting model) [8]. Smart Platform adopts the Publish-and-Subscribe Model for the following reasons:

1) In Publish-and-Subscribe model, agents do not need to hold the prior knowledge of the other party, as did in Direct and Blackboard model, which will help to achieve the loose coupling of the system.

2) In Publish-and-Subscribe model, the communication service primitive is inherently non-blocking, while other models tend to blocking-style primitives, which is preferable for loose coupling, because an agent will not halt for the failure of others.

3) In Publish-and-Subscribe model, agent will be notified the arrival of messages by some kind of callback mechanism, while in Tuple-Space model the agent should keep polling the tuple space to fetch the messages. The callback mechanism will allow the programmer to implement an agent more easily and straightforward than active-polling mechanism.

Message Group

The granularity of subscription still needs further consideration in Publish-and-Subscribe model. Here we introduced the notion of Message Group into Smart Platform. Agents subscribe and publish messages on the basis of Message Groups. Message Groups are created dynamically by agents' request in runtime. By introducing this level of abstraction, we offer the developers the flexibility to choose the granularity at their own will. The two extreme cases are, one message group for all messages and one message group for each message. However, they are not good practices either. For the former case, it is essentially a broadcast model, where the efficiency of the whole system is degraded for agents have to check all the messages that might be irrelevant to them. For the latter case, it aggravates the burden of the communication layer, so it is not scalable. Our experience here is organizing message groups according to topics. For example, all messages related to the environmental context of the room can be grouped into a Message Group called "Room Context".

Besides its name, a message group has another important property - working mode, which could be either message-oriented or stream-oriented. This property determines which communication scheme Smart Platform will use to deliver messages. For message-oriented message groups, the mediated scheme will be used. More specifically, the DS handles all the subscriptions centrally and the messages are conveyed on the established connection between agents and DS. For stream-oriented message groups, the directed scheme will be used. More specifically, first, DS will assign a multicast address to each possible stream-oriented message group in the system. Then when an agent subscribes to such a message group, it has to acquire the multicast-address for this message group from DS first and listens on this address. On the other hand, if an agent wants to publish a message to such a message group, it also has to lookup the DS for the assigned multicast address first and sends the message on this address. For the sake of efficiency, agents will cache the multicast address locally.

Speech Acts

Speech Act theory is often used in the research of MAS to describe the possible intentions of communication efforts initiated by an agent [9]. More than a dozen of speech acts have been proposed in that field, such as inform, reply, tell, ask, achieve, deny. According to our observation, we found it is enough to only distinguish two kinds of speech act in Smart Spaces, and other speech acts can be achieved equivalently by them or their combination. These two basic kinds are

1) Inform. This is a kind of communication efforts where no reply is needed. For example, an agent informs other agents that a specific event happened in the environment or an agent asks other agents to complete a command, but it does not care whether the command will be actually executed or not.

2) Query. This is a kind of communication efforts which asks for a reply. For example an agent asks other agents to return a specific information or an agent asks other agent to complete a specific command and it does care whether the command is successfully executed or not.

The inform speech act can be performed by the Publish-and-Subscribe model naturally. However performing the query speech act is a little difficult in this model (In contrast, the Direct Communication model is most appropriate to achieve this task). Here we enhanced the ordinary Publish-and-Subscribe with a Reply-ID mechanism. Whenever an agent should initiate a query-type communication, it publishes the message as well as a ReplyID, which is essentially a string generated by the agent to identify the message. Any agents who subscribed the message group will receive the message together with the ReplyID. When those agents need to response to the received message, they publish the reply message to a specific system-defined message group along with the received ReplyID. Then the DS will forward the message back to the agent by invoking a callback function of the agent.

OAA uses a "term-unification" model to coordinate the agents, which is borrowed from the Prolog. Agents should first declare what kind of term (named solvable) it can solve. When an agent needs to communicate with others, it sends a term need to solve to a centered component named Facilitator. Then Facilitator will forward the terms to all agents who have declared solvable that can be unified with the requested term, gather their answers, and return them all at once in the answers list back to the initiator. This model is essentially a Publish-and-Subscribe model. However, the subscription granularity is fixed to the basis of individual messages, which is not scalable as we stated above. Event Heap uses a Tuple-Space model. As mentioned above, its main shortcoming is the agent should keep polling the tuple-space, which is not as convenient as callback mechanism in Smart Platform for agent developers (As a compromise, its developers newly also introduced a callback mechanism into Event Heap. However, we think the underlying coordination model of this mechanism actually becomes to a Publish-and-Subscribe model, but no longer a Tuple-Space model). However, Event Heap has one advantage inherited from tuple-space: persistency, i.e., the posted messages on tuple-space will remain in the tuple-space until deleted by an explicit request or after a specified time. This helps a rebooted agent keep up with other agents more easily, for all its needed messages are likely to

be still there. We are thinking about the possibility to integrate this nature into our Publish-and-Subscribe model in Smart Platform by associating a Time-To-Live field to each messages posted. Metagluue provides both the Direct Communication model and Publish-and-Subscribe model. However, since the single enhanced Publish-and-Subscribe model in Smart Platform can afford as much capability as the sum of the two models used in Metagluue, our approach is much preferable because the agent developers only need to grasp one set of communication primitives instead of two.

3.5 Common Services

Currently we built three types of common service into Smart Platform that we think are most needed in Smart Spaces.

Automatic Participating in the Runtime Environment

To enable a computer to participate the runtime environment of Smart Platform, the Container on it should establish a connection and register to the DS. Instead of manually configuring the Container to tell it where the DS is, we implemented a discover-and-join mechanism which allow the Container automatically track the location of the DS and keep the connection with it. The mechanism is implemented as following:

- 1) While running, the DS keeps listening on a predefined multicast address.
- 2) While startup, the Container sends a ping packet to this multicast address.
- 3) Upon receiving the ping packet, the DS will reply with a message containing the TCP port and address where it can be connected
- 4) Upon receiving the reply packet, the Container begins to establish a connection with the DS on the reported TCP address and register to it.
- 5) While the TCP connection with the DS is lost, the Container starts the above procedure again until find the DS.

We found this mechanism very useful in developing period of our Smart Classroom. In order to integrate their computers into the runtime environment, other members only need to download a copy of the Container and run it. No manual configuration is required here. Furthermore the fact that we might move and restart the DS from time to time is totally transparent to them.

As far as we know, none of the other three systems provide similar capability, where system configuration should be either manually configured on the computer or hard-coded in the agent.

Agent-Dependency Management and Resolving

Agents are of dependence on each other. An agent might refuse to work or exhibit different behavior if certain other agents are missing. In Smart Platform we introduce the Service abstraction to help managing the decencies among agents. A dependency between two agents is formed if one agent provides a certain service and the other relies on the same service. Describing the agent dependencies in the Service

abstraction instead of in the agent name directly introduces a level of flexibility and caters for loose-couplings.

While startup, an agent is asked to register the services it can provide and the services it depend on to the DS. Then DS will look up its registration database to check whether all the asked services have already been provided by current agents in the system. For each missing service, the DS will try to locate the agent that can provide it. The knowledge of which agent can provide this service and where its execution code can possibly be found is acquired from the accumulated registration data history in the DS, which records the name of agent ever appeared, the services it can provide, and the host where it last appeared. If the agent is located, DS will contact with the Container on the host it last appeared to launch it. After all these tasks have been done, the initial agent will get a notification (via a callback function) to indicate whether each dependency is satisfied or not, and then the agent could adjust its behavior accordingly.

A by-product of this automatic agent-dependency resolving mechanism is that the whole Smart Space system can be brought into running by launching only several initial agents, as long as developers has carefully designed the dependencies between the agents. We found this feature very useful while debugging our Smart Classroom, because we need to restart the whole system frequently then.

In Metaglu, the dependency between agents and the dependency on physical resources are represented in a unified framework, and a dedicated mechanism called Rascal [10] is developed to handle related management task. OAA and Event Heap do not have the capability to describe and maintain the agent dependency.

Debugging facilities

Traditional debug tools work well to isolate and locate bugs only related to a single process. However, in a Smart Space, more bugs are due to the agent's misbehavior after a certain sequence of interaction with other agents, or when a certain external environment situation exits, including both the states of other agents and the physical environment in a Smart Space. In this case traditional debug tools helps little. We built some integrated mechanism and provide a dedicated agent called Monitor to assist the developers to debug the system in this case. Developers are allowed to accomplish the following tasks with the Monitor. First, developers can get a real-time overall picture of the system. The displayed information includes the computers currently jointing the runtime environment, the agents running on these computers, the status of these agents, the topology of agent-dependencies, and so on. Second, developers can remotely launch or kill an agent. Third, developers can trace out the exchanged messages on the fly. Finally, developers manually compose a message and post it to a message group to simulate the behavior of some agents.

3.6 Wire-protocol based approach to cross-platform and cross- language issue

We tackle the issue of supporting heterogeneous platforms and languages of Smart Platform by means of establishing a set of well-defined and open wire-protocol both between agent and Container and between Container and DS. The wire-protocol

includes the predefined contacting address, the format and semantic of each message and the sequence of the transactions. A process will be considered as a valid agent, Container or DS as long as it complies with the corresponding wire-protocol, no matter what its underlying hardware and software platform are, how it is implemented or what programming language it used. Although we currently only provide Container and DS running on Windows platform and agent development library in C++ and Java, any other researchers can easily provide their own implementations on other platforms and languages according to our open wire-protocol. It is worth to mention that the wire-level message in Smart Platform is also encoded in XML for the advantages of XML.

Compared with our approach, Metaglué depends on the Java language to achieve the same goal, where the core functions of the runtime environment are encoded in Java and remained within the Java VM. We found this approach has several shortcomings. First, the interpreted execution nature of Java (in despite of the invention of JIT) makes the system implemented on it less efficient and thus limited the performance of the SISS. Second, since many real-time processing algorithms are implemented in C or C++ for the sake of performance, developers usually have to use the mechanism of JNI (Java Native Interface) [11] to access the runtime environment of the SISS. However, according to our practice, JNI is quite cumbersome and fragile, resulting the difficulties during the development. Third, since they are closed system, it makes difficult to share efforts with other research groups. OAA is also built on the wire-protocol based approach, but the protocol is not open. The initial version of Event Heap also depends on Java, so it suffers the same difficulties as Metaglué. However, in its most recent version its developers have revised the design to adopt the wire-protocol based approach similar to Smart Platform.

3.7 Agent Development Kit

In order to ease the task of developing agents, Smart Platform provides an Agent Development Kit for agent developers. It is composed of:

- 1) A C++ and Java version of agent development library that encapsulate the implementation of the wire-protocol on the agent side in to a base class – CAgent. The CAgent can be easily used with a set of simple methods or overridables. There are mainly only 5 of them, Register, Subscribe, Publish, OnNotifyMsg and Leave. Their corresponding semantics are clearly reflected by their name, so we will not elaborate on them here.

- 2) A Custom AppWizard in Visual C++ that can automatically generate the skeleton code for a new agent. This utility further ease the efforts required to develop an agent.

- 3) A standard setup program that can automatically copy, configure and run the needed components of Smart Classroom onto a new computer, which greatly reduce the work of deployment.

4 Evaluation of Smart Platform

4.1 Application in Smart Classroom

The effectiveness of the Smart Platform has been validated by its real use in our Smart Classroom project. This project is to develop a Smart Space for Tele-education. In the Smart Classroom, teachers can instruct and interact with remote students just like in an ordinary classroom, which overcomes the shortcoming of traditional desktop-based Tele-education system where the teachers have to remain stationary in front of a desktop computer and use unnatural ways, say, mouse and keyboard to instruct remote students. The involved technologies and modules consist of human and hand tracking, face recognition, speaker recognition, speech recognition, CSCW and so on. They are encapsulated in about eleven agents running on Smart Platform, which are distributed among eight networked computers. Fig. 5 illustrates the constituent agents in Smart Classroom and their dependencies. In its application in Smart Classroom, Smart Platform has exhibited advantages to OAA, which we once had used in the initial stage of the project, in its loose coupling structure, system performance and robustness.

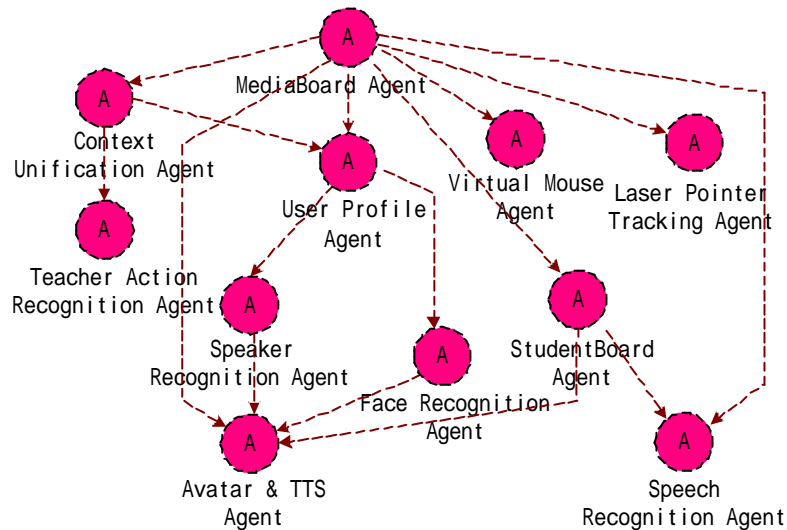


Fig. 5. Agents in Smart Classroom and their dependencies

We also made an informal usability study of Smart Platform while training other members of Smart Classroom project to use it. About seven members accepted the training. They were with different research backgrounds and their experience about distributed computing also ranged from experienced to naive. According to our observation, most of them could understand the principle of Smart Platform and grasped the usage of the ADK in less than one hour. Half of them could deploy Smart

Platform on their own computer and start their agent-development work on it without any further help from us. This proved the ease-of-use of the Smart Platform in some sense.

4.2 Performance Evaluation

We carried out a performance evaluation on Smart Platform to evaluate its throughput and delivery latency. The experiment is conducted on seven networked computers connected by a dedicated 100M Ethernet LAN. DS resides on Computer A. A Ping agent and a Pong agent are running on Computer B and Computer C respectively. These two agents together are used to measure the Round-Trip Time (RTT) of the system by sending and bouncing probing messages on a message group. The probability density of the length of the probing message is a uniform distribution on [0,1000] bytes. All other computers are used to run agents that publish messages in a rate according to Poisson process to simulate the load on Smart Platform. The probability density of the message length is a uniform distribution on [100, 200] bytes. We observed the RTT in different system loads. The result is illustrated in Fig. 6 (a). The experiment shows the RTT grows from 13 ms to 36 ms linearly with the increase of the load of the system from 0 to 960 Messages/S (The delivery latency can be roughly estimated as a half of the RTT). It also shows the maximum throughput of the system is about 960 Messages/S, where the CPU load of the computer where DS is running has reached 100%. The configuration of all the computers in the experiments are PIII 1.6G / 128M RAM / Win2K and the C++ version of agent library is used.

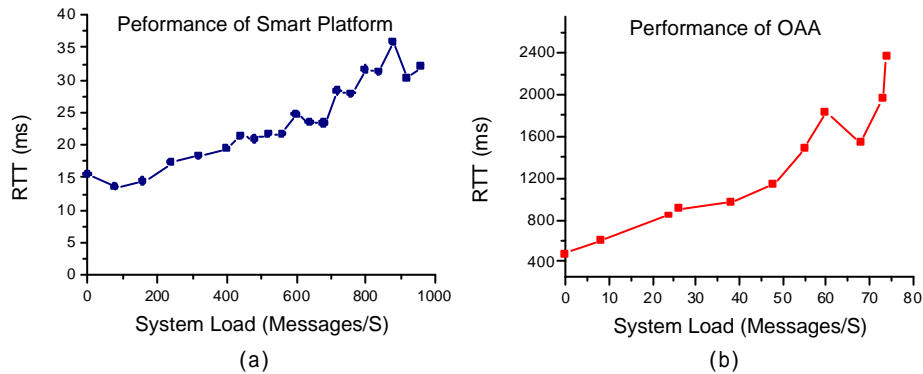


Fig. 6. RTT and throughput performance of Smart Platform and OAA

For comparison, Fig. 6 (b) gives the result of a similar experiment that we conducted on OAA under the same condition and with the same parameters. The result shows the RTT is from 500ms to 2400ms as the system load increasing from 0 to 80 Messages/S. The experiment also shows the bottleneck of OAA does not lie in the Facilitator component, but in the agent library. An agent sending messages at a rate about 40 Messages/S will result in a 100% CPU load. (The agents used in the

experiment are developed with the OAA2.1 C library in MS VC++ 6.0. However, due to some bugs in the provided C library, running the agent built with the Release settings will cause a runtime error. Therefore the agents used in the experiment are all built with the Debug settings, which accounts for the poor performance of OAA to some extent.) The experiment demonstrated that Smart Platform provides an order of magnitude improvement in both the RTT performance and the maximum throughput over OAA.

5 Interoperation of heterogeneous SISSs

Recently we are cooperating with researchers from MIT, AI Lab to interconnect our Smart Classroom with their Intelligent Room in order to explore some more interesting applications for both systems. The issue of interoperation of heterogeneous SISSs rises here, because the Smart Classroom is built on Smart Platform while Intelligent Room is built on Metaglue. We believe this issue will be an important research topic, since more and more Smart Spaces are built by researchers from all over the world and it is a natural demand that these individual spaces could be assembled or have some type of interactions. As a result, it is inevitable to deal with heterogeneous SISSs.

We present a dual-citizenship-agent based approach to this issue. The philosophy here is that, instead of modifying the system structure of either of the SISSs, a special dual-citizenship agent, which comply with the specifications of both SISSs, is developed and placed between the two SISSs to function as a proxy to each other side. While bridges the gap between the two SISSs, this approach remains the autonomy of both sides, for all other agents do not need to know anything about the agents in the other side.

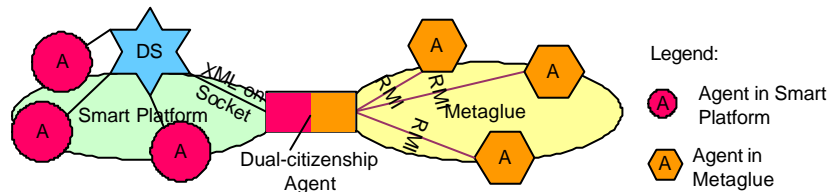


Fig. 7. The proposed approach to interoperate Smart Platform and Metaglue

Fig. 7 illustrates the presented structure for the interoperation between Smart Platform and Metaglue. We use an example to depict how the interaction of the two rooms is achieved by this structure. A sample scenario proposed here is the occupants in one room could be aware of persons' coming and leaving in the other room. This example is implemented in the following way. When a user comes into Smart Classroom, the UserIdentify agent in Smart Classroom will acquire the user's ID through a combination of face-recognition and speaker-recognition and publish a message for this event to the RoomRelatedMsgs message group. Then the dual-citizenship agent will be notified by this message since it has subscribed the RoomRelatedMsgs message group, and it will use the communication mechanism in Metaglue, RMI, to invoke another agent in Intelligent Room to notify the occupants there.

The key problem here is how to make the implementation of the dual-citizenship agent independent of the specific applications. We are currently investigating a declarative way to separate the knowledge of how to map messages for the two sides, which is application-dependent, from other application-independent logic.

6 Conclusions and Future Work

A Software Infrastructure for Smart Space (SISS) provides the runtime environment and common services for the distributed modules in a Smart Space to connect, communicate and collaborate with each other. The distinctive nature of Smart Spaces imposes some inherent requirements on its underlying SISS. Due to its loose coupling structure, XML-based message syntax, Publish-and-Subscribe coordination model, hybrid communication scheme, as well as other important mechanisms, Smart Platform provides better performance, usability, extensibility and interoperability, compared with other prior systems. As one of our initial motivation, we have also made it an open solution available for other researchers.

As the next step, we are considering to study the issues of service discovery, resource management and context-awareness computing in the Smart Space and to build the support for them as an integrated part of Smart Platform.

Acknowledgement

Graduate students who have been involved in project Smart Platform and Smart Classroom include Changhao Jiang (now a research scientist at CMU working for project Aura), Enyi Chen, Zhongnan Shen, Yi Che, Yunzhang Pei (now a research scientist at IBM CRL), Kun Tan (now a research scientist at Microsoft Research China), Chunyuan Liao (now a PhD student at University of Maryland), Haibing Ren, Fang Liu, Qiang Wang, Gang Song and Zhiyuan He. To connect the two Smart Spaces, Prof. Trevor Darrell, Mr. Krzysztof Gajos, Dr. Howard Shrore, Frank R. Bentley and Stefanie Chiou at MIT AI lab are now doing effort to make it come true. This project is supported by NSFC (National Science Foundation of China), National High-tech Developing Plan, High Level Research Plan of Tsinghua University and IBM-Tsinghua Innovation Institute Grants.

References

1. Michael Coen, Brenton Phillips, Nimrod Warshawsky, et al. Meeting the computational needs of intelligent environments: The Metaglove system. In Proceedings of MANSE'99, Dublin, Ireland, 1999.

2. Barry Brumitt, Brian Meyers, John Krumm, et al. EasyLiving: Technologies for intelligent environments. In Proceedings of Second International Symposium on Handheld and Ubiquitous Computing (HUC 2000), 12-29, Bristol, UK. Springer, LNCS1927.
3. Martin, D., Cheyer, A. & Moran, D.. The Open Agent Architecture: A framework for building distributed software systems. Applied Artificial Intelligence: An International Journal. 91-128, Vol.13, No.1-2. 1999
4. Armando Fox, Brad Johanson, Pat Hanrahan and Terry Winograd. Integrating Information Appliances into an Interactive Workspace. IEEE Computer Graphics and Applications, 54-65, Vol. 20, No. 3, 2000
5. Weikai Xie, Yuanchun Shi and Guanyou Xu. Smart Classroom - an Intelligent Environment for Tele-education. In Proceedings of The Second Pacific-Rim Conference on Multimedia (PCM 2001), 662-668, Beijing, China. Springer LNCS2195
6. http://media.cs.tsinghua.edu.cn/smart_platform
7. <http://www.ietf.org/rfc/rfc1889.txt>
8. Cabri, L. Leonardi, F. Zambonelli, "How to Coordinate Internet Applications based on Mobile Agents", In Proceedings of IEEE Seventh International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 1998), 104-109, Stanford, USA, IEEE CS Press
9. Yannis Labrou, Tim Finin, Yun Peng. Agent Communication Languages: The Current Landscape. IEEE Intelligent Systems, 45-52, March/April, 1999
10. Krzysztof Gajos. Rascal - a resource manager for multi agent systems in smart spaces. In Proceedings of CEEMAS 2001, 111-120, Springer LNAI2296.
11. <http://java.sun.com/products/jdk/1.2/docs/guide/jni/>