SMART PLATFORM DEVELOPER’S GUIDE

1 INTRODUCTION

2 THE FEATURES AND DESIGN OBJECTIVES OF SMART PLATFORM

2.1 Dynamically Find and Join

2.2 The combination of Delegated Communication (Message oriented) and Peer to Peer Communication (Stream-oriented)

2.3 Automatically Manage and Resolve of the Agent Dependencies

2.4 XML based ICL (Inter-agent Communication Language)

3 ARCHITECTURE

4 THE DS

4.1 The Startup of DS

4.2 Alter the default listening address of DS (Setup two separated Smart Platforms environments on a LAN)

4.3 Internal Structure of DS

4.4 The Agent Registers to the Directory Service

4.5 Message Dispatcher

4.5.1 The Message Group

4.5.2 The built-in message group----SysEvent (System Event)

4.5.3 Message Format

4.6 The Dependency Manager

4.6.1 Three Kinds of Status of Agent

5 THE CONTAINER

5.1 The Startup of a Container

5.2 Container Find and Join the DS

5.3 Agent Joins the Smart Platform via Container

5.4 Configure Your Container

5.5 The GUI of Container

5.6 Kill an Agent

5.7 Start a Previously Running Agent

6 STREAM-ORIENTED COMMUNICATION

6.1 How to use the stream-oriented communication

7 MONITOR AGENT

8 INSTALL AND RUN SMART PLATFORM

8.1 Installation of Smart Platform

8.2 How to Run Smart Platform

9 THE ADK (AGENT DEVELOPMENT KIT)

9.1 The VC++ version ADK

9.2 Smart Platform Application Wizard
1 Introduction

As an important notion in the vision of Ubiquitous Computing, Smart Space or Intelligent Environment is defined as an enhanced a physical space where people can get the services of computer systems without approaching computers or using the cumbersome keyboard or mouse to interact with them. Typically a Smart Space is a distributed system that involves many sensors, perception devices, software modules and computers. To develop and support such a complicated system, some type of software platform is a must.

Smart Platform is just designed and developed as the Software Infrastructure for Smart Space (SISS). This Developer’s Guide introduces the features and the architecture of Smart Platform, and provides a description of how to develop an agent on this platform.

We assume general familiarity with software engineering practices and programming language concepts. Some knowledge about the basics of the XML language is also helpful, as XML serves as the basis communication language throughout the whole platform.

2 The Features and Design Objectives of Smart Platform

Smart Platform is modeled as a Multi-Agent System, in which the basic unit is agent. Each agent is an autonomous process that either contributes some services to the whole system or uses the service of other agents to achieve a specific goal. As a software platform, Smart Platform implements a runtime environment for the agents and provides an agent developing kit for the developers, including a programming interface and some debug tools.

To accommodate the characteristics of Smart Space system, Smart Platform is designed with the following features.

2.1 Dynamically Find and Join

In the traditional way, if we want to access a service on the network, we must have prior knowledge of the access address of the service, i.e. the IP address of the server and the port of the service on the server. While typically there will be dozens of agents in a decent Smart Space system, it is so tedious if we have to manually interconnect every agent. Instead, with the “find and join” mechanism, Smart Platform uses IP multicasting to dynamically discover and assemble the computation environment, eliminating the need of manual configuration.
2.2 The combination of Delegated Communication (Message oriented) and Peer to Peer Communication (Stream-oriented)

The Smart Platform provides two different communication modes, “Delegated” and “Peer to Peer”. In “Delegated” mode, all messages between agents are mediated by a special runtime module called Message Dispatcher (chapter 4.5), which is a subcomponent of DS (chapter 4). Agents post and accept messages without knowledge about the identity of the other party. It is the responsibility of Message Dispatcher to forward the messages to the proper agent according to the contents of the message. Thus, agents are loosely coupled. This category of commutation mechanism is aimed for messages that occasionally occur and usually with high-level semantics. This communication mode is also called message-oriented communication.

However, in the case of inter-agent data exchange in high volumes, e.g. the transportation of real-time media, this mode of communication is not readily applicable, for the overburdened DS is likely to become a potential bottleneck. Fortunately, the Smart Platform provides a second category of communication mechanism, namely, the peer-to-peer communication. In this mode, a direct path is established between involving agents, who could then exchange data without having to use the Message Dispatcher as a broker. Our implementation of the peer-to-peer mechanism is marked by two features, support for one-to-many commutation, and support for real-time streaming data (section 6). The former is achieved through IP multicast, and the latter through the underlying Real-time Transport Protocol (RTP RFC1889). Note that Directory Service, another subcomponent of DS, also plays an important role in that. It helps the two agents to negotiate and maintain the communication channel.

Agents are free to select the proper communication mode according to their specific requirements.

2.3 Automatically Manage and Resolve of the Agent Dependencies

In a multi-agent system, agents must collaborate with each other. One agent may use the services provided by another agent. We call this relationship as agent dependence. The topological structure of the dependencies between agents may be a tree or even a net. The Smart Platform can manage and resolve those dependencies.

When each agent joins the computation environment, it must announce the services it provides and the services it depends upon. Smart Platform will store this information in a persistent storage. If an agent starts up asking for a service and the service providing agent happens not to be running, the Smart Platform will use the stored knowledge to locate and automatically launch the agent. This feature is called “Agent Dependency Resolution”.

With this mechanism, it is no longer needed to manually start all the agents in a Smart Space system. Just start some core agents in a system; the whole system can be put into a determinate state.
2.4 XML based ICL (Inter-agent Communication Language)

ICL is the definition of the syntactic structure, and the semantic to some extent, of the messages between agents. We choose the XML as the basis of the syntax of the ICL in Smart Platform. The inherent advantages of XML benefit the Smart Platform in some aspects. (1) The extensibility and its ability to flexibly describe almost all kinds of data makes the ICL can be easily extended. (2) As the one of standard of technology of Internet, it eases the inter-operation of Smart Platform with other heterogeneous systems. (3) There are many software libraries for the processing of XML in both industry and academe, which ease the development of the Smart Platform.

3 Architecture

Smart Platform runs upon networks-connected computers in a Smart Space. Smart Platform masks the boundary of the involved computers and provides a uniform running environment and highly structured communication model for the software modules run on the platform.

The runtime environment is composed of three kinds of components, which are Agent, Container and DS.

1) An Agent is the basic encapsulation of the software modules of the systems.
2) Each computer participating in the runtime environment will host a dedicated process called Container, which maintains the agents that run on the same computer and provides system-level services for them. In some sense, Container is the mediator between agent and DS. It makes the low-level communicate details transparent to agent developers and provides a simple communication interface for agent.
3) There is one global dedicated process called DS in the environment. It is the service provider of the environment.

Figure 1 shows the architecture of the whole Smart Platform.

Later in this document we will talk about the DS, the Container, a specially agent ---- the Monitor and the ADK (Agent Development Kits) and the two communication model of Smart Platform.

Fig.1 The architecture of Smart Platform
4 The DS

4.1 The Startup of DS

Normally there should be only one DS runs on a Smart Platform; otherwise, two DS on a same platform may bring Smart Platform into an indeterminate state. Therefore, we build into DS a mechanism to ensure there is only globally one instance of DS running in the system.

1) When a DS is started, it tries to locate a DS on the LAN using UDP multicast, just like a Container does.

2) If (1) fails, it means no other instance of DS runs on the LAN, then it starts to listen to its multicast address and wait for Container to connect.

3) If (1) succeeds, it means there is already a DS runs on the LAN, then this DS informs the user and terminates itself.

4.2 Alter the default listening address of DS (Setup two separated Smart Platforms environments on a LAN)

Although as mentioned above, Smart Platform prevents running more than one instance of DS on a LAN by default, we do provide a mechanism to do so by manually altering the default listening port and address of the DS. Consider the following case:

Group A and Group B are working on two completely different projects, they are on the same LAN and they both use the Smart Platform. If they have to share a single DS, there will be some intervention between their work, especially in the developing and debugging phase. Therefore, it makes sense if we can setup two or even more Smart Platform on a same LAN, running without intervention with each other. This is achieved by configure DSs to listen on different addresses.

When the DS is started, it loads the configuration file (Config.xml) in its working directory. The Config.xml file describes the listening multicast address (a D class IP address) of the DS. If there is no such a file or the file format is wrong, the DS will use the default address (234.5.6.7). The following is a sample of the Config.xml file:

```xml
<Config>
    <DSConfig>
        <DS_Multicast_Address>234.5.6.7</DS_Multicast_Address>
    </DSConfig>
    <ContainerConfig>
        <AutoReTryInterval>30</AutoReTryInterval>
    </ContainerConfig>
</Config>
```

Fig. 2 Sample of Config.xml

The <DS_Multicast_Address> element describes the Multicast IP on which the DS listens, this must be a D class IP, or else the DS may fail to listen to the address. The <ContainerConfig>
element is intended for the Container, so we do not care about it here.

4.3 Internal Structure of DS

The structure of the DS can be divided into three subcomponents, each provides a different set of services, as illustrated in Figure 1.

1) The Directory Services subcomponent provides services such as agent register and query. When an agent is started, it must register its basic information to Directory Service, such as its name, the services it provides, the services it depends as well as its development language. The Directory Service provides the directory query service too, that is to say, agents can query it on which host the agent provides the specified service runs.

2) The Message Dispatcher subcomponent implements the message oriented communication service for the agents in Smart Platform. We will describe this component in detail later in chapter 4.5.

3) The Dependency Manager subcomponent is responsible for the management and maintenance of the agent dependencies. We will also describe this component in detail later in chapter 4.6.

4.4 The Agent Registers to the Directory Service

When an agent is started, it connects to the local container in the first step. Then the agent should register itself to Directory Service so it can join Smart Platform. It sends its registration information through the container to DS. This information includes its name, develop language, the name of the container, the services it provides as well as the services it depends. When receiving those information, the Directory Service will store the name of the agent, the name of the container and the services this agent provides into the Directory Service database. This database will be used by the Dependency Manager to find out agents to satisfy the inter-agent dependencies. If the dependency of the agents is not null (i.e it depends on some other service), the Directory Service notifies the Dependency Manager to check the dependencies. After that, the Directory Service replies a message to the container saying the agent has successfully registered to the Directory Service and the container will notify the agent in turn.

4.5 Message Dispatcher

Message Dispatcher plays a key role in the “Delegated Communication” mode. Every agent can publish some kind of messages on Message Dispatcher; it can also subscribe the desired message from Message Dispatcher. Message Dispatcher is responsible for forwarding the messages to the appropriate agents by the notion of Message Group as described below. In addition, it should also check the validity of a message when an agent publishes it. If it is invalid, it just ignores the message. (This function hasn’t been implemented in this version) Delegated Communication mode is ideal for most inter-agent communications for its inherent advantage of loose coupling.
4.5.1 The Message Group

The message group is a key notion in the Delegated Communication mode. All the defined inter-agent messages are grouped into message groups, which make a logical point where agents can publish message into and subscribe messages from. The messages in a message group usually have more commonalities than the messages in other groups. For example, all the messages about the control of the cameras may be grouped into a CameraControl message group. There may be many message groups on Message Dispatcher. A message in a group can describe either an event or a command. For example, a camera agent may publishes a message to the PersonContext group when it detects a person comes into the room from the door. It is also possible for a TTS (Text To Speech) agent subscribe the ArtificialVoice group. When an agent wants the TTS agent to read a short sentence out, it may simply publish a message in this group.

If an agent is interested in a certain message group, it may subscribe this message group. When a message of this group arrives, the Message Dispatcher will forward this message to the agent, and typically the agent will response by invoke a message processing method. (We will discuss the details later in ADK in chapter 8)

By the notion of Message Group, the agent can communicate with each other without the prior knowledge who is the other part. All they need to care is the message and the proper message group itself.

4.5.2 The built-in message group----SysEvent (System Event)

SysEvent is a built-in message group on the Facilitator. The messages in the SysEvent group describe the status change of the Smart Platform. Only the Message Dispatcher can publish SysEvent messages. A system event message may describe one of the following events.

a) A Container joins or leaves the Smart Platform.
b) An Agent joins or leaves the Smart Platform.
c) An Agent subscribes or unsubscribes a message group.
d) The status of the agent changes (e.g. from unsatisfied to satisfied).

4.5.3 Message Format

The word “message” in Smart Platform has different meaning for the agent developers and the Smart Platform itself. We call them user-level messages and low-level messages respectively.

User-level message is what the agent developers see. While a user-level message is published and transported in Smart Platform, it will be encapsulated into a low-level message, which is what is seen by the runtime entities in Smart Platform. Put it another way, the low-level message is transmitted from the Container to Message Dispatcher then from Message Dispatcher to the container where the subscriber is in. Then container extract the encapsulated user-level message from the low-level message and passes it to the agent.

A user message in the Smart Platform must be a valid XML document. For example:

```xml
<SysEvent>
  <QueryType>New-Agent</QueryType>
</SysEvent>
```
Fig. 3 An example of a user-level message

The syntax of Low-level messages is also based on XML. Following is the format of the low-level message corresponding to the user-level message showed in Figure 3:

```xml
<Msg>
  <Command NeedReply="No">Publish</Command>
  <Sender AgtName="Blackboard" CtnID="0" AgtID="0" CtnName="Directory Services" />
  <Groups>
    <Group Name="SysEvent" />
  </Groups>
  <Agents />
  <Receivers>
  </Receivers>
  <Contents>
    <SysEvent>
      <QueryType>New-Agent</QueryType>
      <Agent>
        <Name>Monitor alfa 1.0</Name>
        <CtnID>1</CtnID>
        <ID>2</ID>
        <DevLanguage>Visual C++ 6.0</DevLanguage>
        <Services />
        <Dependencies />
        <Status>Satisfied</Status>
      </Agent>
    </SysEvent>
  </Contents>
</Msg>
```

Fig. 4 An example of a low-level message

Besides the user level message that is contained in the `<Contents>` element, there are also several other fields of the low-level message are visible to the agent developers. They are the name of the agent that publishes the message as the `AgtName` attribute of the `<Sender>` element.
indicate, the name of the Container where the publisher is as the *CtnName* attribute of the `<Sender>` element indicate and the `<ReplyTag>` element.

*<ReplyTag>* is used to implement the query-and-return type of communication. When an agent expects it’s a query message to be replied, it will specify a ReplyTag while publish the message. When a appropriate agent wants to answer the query, it will set the InReplyTo parameter to the same value as the ReplyTag value of the received message while publish the reply message. The when the original agent gets the reply, it may know which message the current received message replies to.

### 4.6 The Dependency Manager

The Dependency Manager automatically maintains and resolves the dependencies of the running agents.

When a new agent joins in, the Dependency Manager checks whether the dependencies of this agent can be satisfied by the currently running agent. If the dependencies cannot be satisfied, the Facilitator will try to start some agents to satisfy them according to the information of the previous running agent.

Every time an agent joins or leaves the Smart Platform, Dependency Manager will check whether the dependencies of other agents will be affected. If that’s the case, the Facilitator will change the affected agents’ status and inform the agent of the status change.

#### 4.6.1 Three Kinds of Status of Agent

As mentioned above, there are three kind of status of an agent: *Satisfied*, *Satisfying* and *Unsatisfied*. They are described as following.

- **Satisfied**: All the dependencies of this agent are satisfied by currently running agents.
- **Satisfying**: Message Dispatcher is now trying to satisfy the agent’s dependencies.
- **Unsatisfied**: The dependencies of this agent cannot be satisfied.

### 5 The Container

In Smart Platform, a special daemon process should run on every host that participates in the runtime environment. It behaviors like an agency between agents on the same computer and the DS, we call this process Container. It communicates with the DS, and provides the Smart Platform system services for the agents that run on the same host. For example, agents can query other agent’s status via the Container, agents can register itself to the DS via the Container, and agents can subscribe or publish message via the Container. All the agents only communicate with the container on its local host, all the low-level communications of the Smart Platform is accomplished and masked by Container.
5.1 The Startup of a Container

There should be no more than one container runs on a host, so when a container is started, it will check if there is another instance of container is now running on the same host. If it is the case, it will notify the user and terminates itself.

Then the container tries to find the address of the DS and join itself to Smart Platform. The procedure of the container joins the DS will be discussed in chapter 5.2.

If the registration fails, the Container will notify the user that no DS can be found in the LAN, the user may choose to let the container retry immediately, try again later or quit immediately. We will discuss more details on it in later chapter 5.4.

After the Container successfully registers itself to the DS, the Container will listen on the pre-defined TCP port 1234 of localhost to wait for the agents’ startup.

5.2 Container Find and Join the DS

Whenever a container starts up, it will locate the DS and connect to it. As the DS is ready, it will listen on a port of a pre-configured multicast address. When a Container is started, first, it sends a UDP multicast packet to that multicast address to discover the DS. When DS receives this packet, it will multicast its host name (IP) and its TCP listening port, so that after the container receives them, it can establish a reliable TCP connection to the DS. As soon as the connection is established, the container registers its information to the Directory Services on the DS. As we have assumed the Smart Platform runs on a LAN and there should be no more than one Container run on a host, so we just use the host name to identify a Container. The Directory Service also assigns a unique ID to the container, which is used for low-level communication between the DS and the Container.

5.3 Agent Joins the Smart Platform via Container

All the agents communicate with the Container via local TCP connection.

When an agent is started, it tries to connect the TCP port 1234 on the same host, where the container should be listening. Once connected, the Container will assign a unique agent ID to identify the agent. Then the agent try to register itself to the Smart Platform, it sends a message to the DS via the Container. We have discussed the procedure in previous chapter 4.4.

In addition, the agent also sends the full path of its executable file to the Container. The container stores the name of the agent and the path of it to a persistent storage (actually the system registry in Windows platform), so that when the container wants to start an agent once running on the host, it can use the stored information to find out where the executable file is and launches it when necessary.

5.4 Configure Your Container

As has been discussed in chapter 4.2, user can configure the multicast address of the DS, so that more than one Smart Platform can run on the same LAN. When there are several DSs run on a
LAN, the user can also configure the Container and tell it which DS to join.

Just like the DS, the container uses the configuration file (Config.xml) from its working directory to learn the **DS Multicast IP**. The format of the Config.xml has been showed in Figure 2. The `<DS_Multicast_Address>` element describes the **DS Multicast IP**.

If there is no such a Config.xml file or the file format is wrong, the Container will use the default address (234.5.6.7).

There is also a `<ContainerConfig>` element in the Config.xml. Its child element, `<AutoReTryInterval>`, shows the interval (valued in seconds) that the container wait before automatically retry to connect to the DS, when it loses the connection with the DS or when it can not find the DS while startup and the user let it to retry later. If this value is non-zero and the connection between the DS and the Container do not exist, the container will try to find and join the DS every `AutoReTryInterval` seconds, until the connection is re-established or the user terminates it.

If there is no such a Config.xml file or the file format is wrong, the interval is set to zero, which means when the connection is lost, the Container will notify the user of this event. It is up to the user to choose to terminate the container or try to connect to the DS immediately.

If the DS is not ready when a container is started, the container will not be able to find and join the Smart Platform. Then the container will notify the user that no DS is not found on LAN, and let the user choose what to do next. The choices are different depends on whether the `AutoReTryInterval` is zero or not. When the interval is not zero, three choices are available, they are “Yes”, “No” and “Cancel”. When the interval is zero, only “Yes” and “No” are available. The following is the meaning of these three choices:

<table>
<thead>
<tr>
<th>Choice</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Try to find and join the DS again immediately.</td>
</tr>
<tr>
<td>No</td>
<td>Terminate itself.</td>
</tr>
<tr>
<td>Cancel</td>
<td>Try to find and join the DS after every <code>AutoReTryInterval</code> second.</td>
</tr>
</tbody>
</table>

**Fig. 5** The Container can’t find the DS
5.5 The GUI of Container

The Fig.6 shows the GUI of the container. There are five areas on the interface, they shows the status and information of the container and some information about the DS.

Area 1 gives the container’s basic information: the name of the container (that is the name of the host) and the development language of the container (in current, we only have VC++ version).

Area 2 gives the DS’s basic information, if the container successfully connect to the DS, the name (IP) and the port of the DS will be showed here.

Area 3 shows all the currently running agents on this container.

Area 4 shows the information of a particular agent that is selected in area 3. The information includes the name of the agent, the develop language of the agent, the services it provides and the services it depends.

Area 5 shows the status of the container or the latest event that takes place in this container.

5.6 Kill an Agent

Container allows the user to kill any agent managed by it. First select the agent you want to kill in the agents’ list box (area 3), click the “Kill Agent” button (Button 6 in Fig.6). Then the container will send a message to the agent to inform the agent to quit. As a programming convention, the agent should terminate itself, as soon as it receives this message.
5.7 Start a Previously Running Agent

Container provides a helpful shortcut to run an agent, which has once run on the computer managed by the container. As the container has stored the agent name with its path in a persistent storage, it is not necessary for you to find the executable file of the agent and launch it with the operating system command whenever you want to start this agent again. Just click the “Start Agent” button (7 in Fig.6), the Container will lookup the storage, and check if the agents’ executable file recorded in the storage is still available (Container will delete the items that are not available now automatically). Then a dialog will pop up. All the agents available are listed in the dialog with its path. User can select one and launch it.

The agent information in the persistent storage is also used in the agent dependency resolution procedure. Whenever the DS asks the Container to do so, the container will look up the system registry, find the path of the agent desired, and launch it.

6 Stream-oriented Communication

If an agent wants to deliver real-time streaming data, e.g. live video or speech, to one or more peers, it is possible and highly recommended to build your application on our embedded feature of stream-oriented communication. Because its implementation takes advantage of IP multicast and the Real-time Transport Protocol (RTP), it supports one-to-many communication, and provides services such as timestamping, sequence numbering, which are inherent of RTP. Sequence numbers are necessary to handle the problem of out-of-order packets and lost packets as exposed by UDP, and the timestamp is for the receiver to reconstruct the original timing before playing back the data, so that jitter could be minimized.

Due to the nature of most realtime-streaming applications, a minor degree of packet loss could be harmlessly tolerated, but variation of delivery latency is detrimental. The design of the Smart Platform’s stream-oriented communication mechanism is precisely based upon this notion. Assume the sender sends out data d1,d2,d3,d4 at time t, t+Δt₁,t+Δt₂, t+Δt₃ respectively. The payload will be timestamped so that at the receiving side, a callback function will be invoked at T, T+Δt₂,T+Δt₃, T+Δt₄, with the matching data as its parameter. In case d2 is lost during transmission, the callback function will be invoked at T, T+Δt₁, T+Δt₄.

Two more things have to be noted about the implementation of the stream-oriented communication mechanism.

First, the implementation is multithreaded where we have a listening thread snooping on incoming data packets, and a separate thread is created for each call-back function invoked. Developers must therefore take into account synchronization problems as exposed by all multithread applications.

Second, we give a brief description of the buffering strategy. In order to avoid jitter in data transfer, a certain number of data packets have to be buffered before playback. In case the callback rate has outpaced the data transfer rate during playback, leaving the buffer empty, the callback
function will be held until a certain number of packets have refilled the buffer. The strategy could be outlined by the following chart involving two states and several transitions. The waiting state is where the receiver callback is suppressed, whereas the active state indicates the callback is in function. The minimum number of packets which activates the receiver in a waiting state is a parameter which users could designate according to the requirement of different applications.

![State Diagram]

6.1 How to use the stream-oriented communication

The stream-oriented communication is organized by streaming-message group, which is an abstraction very similar to the message group, which we have mentioned in chapter 4.5.1. To simplify our communication model, we assume that for each stream-oriented message group there should be at most one sender at a time. However multiple receivers is supported by RTP (Real-time transport protocol, RFC 1889), which is the basis of the stream-oriented communication.

To use the stream-oriented communication, the sender peer should call the CreateStreamMessageGroup method of CAgent class, so as to join the multicast group, which is needed for RTP. The receiver peer should call the SubscribeStreamGroup method of CAgent class to join the multicast group and keep listening. In Smart Platform each stream-oriented message group has its unique multicast group, which is assigned by the Directory Service. The D class IP addresses of all the groups are the same to that of the DS, which is motioned in chapter 4.2. We use port number to identify the multicast group; every multicast group has a unique port number pair assigned by the Directory Service (one port number for RTP and another for RTCP). The stream-oriented communication of Smart Platform is so organized in order to allow agents loosely couple, which means as long as the sender peer and the receiver peer joins the multicast group, the communication channel is set up regardless who starts first.

The sender peer can use PublishStreamData method of CAgent to publish stream-orient data regardless whether the receiver has joined the multicast group or not. If it happens that the receiver peer is kept listening at this moment, the data is buffered and called back according to the timestamp attached with the data. Refer to the ADK Reference for detailed information.

LeaveStreamGroup method is called when the sender peer leaves the multicast group and UnsubscribeStreamGroup method is used for the receiver peer to leave. SetStreamBufVolume is used to set the buffer volume at the receiver peer.
7 Monitor Agent

The Monitor agent is a special agent that serves as the watcher and the debugger of the Smart Platform, the Figure above shows a picture of the Monitor’s GUI. There are total four areas on the GUI of the Monitor.

Area 1, which is the tree view of the monitor shows the basic information and status of the Smart Platform. The sub-folders of the Smart Platform folder of the tree indicate the containers that participate in the Smart Platform, while the sub-node of the Container shows the agents that run on the same host as the Container. The small icon of before the name of the agent gives the status of the agent. Sun(☀) means Satisfied, cloudy(☁) means Satisfying while dark clouds(쏘) means Unsatisfied.

User can use the Monitor agent to monitor any agents or message groups in the Smart Platform for debugging purpose. The monitored folder shows the currently monitored agents or message groups. All the monitored messages will be list in corresponding sub-folders.

The Area 2 gives the status of the Blackboard. You can find out how many message groups are registered on the Blackboard and all the subscribers of a desired message group.

Area 3 shows the dependencies of the currently selected agent in the left tree view. Form the right to the left, the illustrate gives the following information: the agent who use the services that provides by the selected agent, the services that provides by the selected agent, the select agent, the services that the selected agent depends on (white means this dependencies is currently satisfied while red means not) and the agents who provide services for the selected agent.

Area 4 gives the detail information of the selected item in the left view. If the selected item is
a folder of a monitored message group or agent, the monitored messages will be list in the list box. Double click the message you can see the contents of the message.

The right button context menu is also very useful in the GUI of the Monitor agent, you can achieve many functions list below of the Monitor agent use this menu, all these functions can ease the debugging of the agent.

- Monitor or un-monitor a select agent or group
- Kill a selected agent
- Start a agent on the selected container
- Send a message to the selected message group

8 Install and Run Smart Platform

8.1 Installation of Smart Platform

![Image of installation process](image)

Fig. 8 A snap during the installation of Smart platform

The above picture shows an image of the installation of Smart Platform. Following the InstallShield Wizard, you can easily install Smart Platform to your computer. The Smart Platform is a completely free to non-commercial purpose, so any serial number can be accepted by the user information dialog during the installation.

8.2 How to Run Smart Platform.

After the installation of Smart Platform, a “Smart Platform” folder is created in the Programs folder of startup menu of Windows. The shortcut or Smart Platform runtimes, such as the DS, the Container and the Montior is placed in this folder. You can easily launch the necessary component of Smart Platform from the startup menu. Figure 9 shows this.
9 The ADK (Agent Development Kit)

The developers of the Smart Space system use ADK to develop agents that can be run on the Smart Platform. ADK is provided in the form class library, in which several core base classes are implemented. Through using and extending these classes, developers can rapidly build agents without the need to know the low-level details of the Smart Platform. We currently developed both VC++ version and java version of the ADK. The VC++ version is packaged in both a static library and dynamic link library form and the Java version is package in a JAR file form.

There are totally four classes encapsulated in the ADK. However, in most case, agent developers only have to know two of them, the CAgent class and the TXML class. These two classes expose basic agent services to the developers, while others implement the low-level details of the communication in the Smart Platform. Only those developers have special purpose or permission need to take care of those low-level classes.

Now we will talk about how to develop agent using the VC++ version ADK. Developers can refer to the ADK VC++ Version Reference for more details.

9.1 The VC++ version ADK

The current VC++ version of ADK includes 7 files:
ADK.h    the C++ header file
ADKD.lib   the debug version of the static link library of ADK
ADKR.lib    the release version of the static link library of ADK
ADKDLLD.lib static link library used for DLL Debug version of ADK
ADKDLLR.lib static link library used for DLL Release version of ADK
ADKDLLD.dll the Debug version dynamic link library of ADK
ADKDLLR.dll the Release version dynamic link library of ADK

We just assume you have installed the Smart Platform to <SPDIR>. The ADK.h can be located in <SPDIR>\Include directory while others can be located in the <SPDIR>\Lib directory. To use the ADK, developer should follow the following instruction.

**Step 1: Import the ADK Static Library**

Import a proper version of ADK static link library to your VC++ project. You can achieve this by configure the settings in the VC++ IDE Menu: Project ->Settings->Link. The following figure shows you how to choose the link library when doing the settings.

<table>
<thead>
<tr>
<th>Version</th>
<th>Library Type</th>
<th>Debug</th>
<th>Release</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static Library</td>
<td>ADKD.LIB</td>
<td>ADKR.LIB</td>
<td></td>
</tr>
<tr>
<td>Dynamic Library</td>
<td>ADKDLLD.LIB</td>
<td>ADKDLLR.LIB</td>
<td></td>
</tr>
</tbody>
</table>

Figure 11. Choose appropriate library file according to difference library type and configuration.

**Step 2: Include the Header file When Necessary**

Include the ADK.h header file, where you want to use the CAgent class or the TXML class. Note that you should define macro ADKDLL if you choose the dynamic link library so as to import the CAgent and TXML class from the dynamic library. You can add “#define ADKDLL” before everywhere you include ADK.h. You can also add ADKDLL to the project settings as a global macro so as to achieve the importation. Thus add ADKDLL to VC++ IDE Menu: Project ->Settings->Resources->Preprocessor definitions is OK.

**Step 3: Initialize and Uninitialize**

As the ADK used the COM to parse the XML document, the agent program should initialize the COM, before it use the ADK. It is also necessary to clean the COM when the program terminates, Note that the stream-oriented communication is implemented with multiple thread. When exiting he main thread should wait until other thread exit, or it may lead to memory leak problem. The following code segment shows an example of how to initialize and uninitialize.

```c++
BOOL CAgentApp::InitInstance()
{
    ....
    CAgentDlg dlg;
    m_pMainWnd = &dlg;
    CoInitialize(NULL); //Initialize the COM
    int nResponse = dlg.DoModal();
    if (nResponse == IDOK)
    {
        // TODO: Place code here to handle when the dialog is dismissed with OK
    }
}
```
Step 4: Inherit the CAgent class.

For example, suppose you want developed a new agent called the TestAgent. You may define a new class CTestAgent who inherits the CAgent class. You should specify the Agent’s name, dependencies, provided services, develop language and other information when you call the constructor of the CAgent class.

Step 5: Overload some virtual methods of the CAgent class.

Three methods can be overloaded if necessary, they are:

```cpp
virtual void OnDisconnected();
virtual void OnDependencySatisfied();
virtual void OnDependLost(CString strLeavingAgtName, CString strDepname);
```

You can refer to the ADK VC++ Version Reference for detailed information.

Step 6: Call the Methods of CAgent Class: Register and Subscribe.

An agent should call the Register method at the very beginning of its initialization, in order to register to the DS and participate the Smart Platform environment.

```cpp
BOOL Register();
```

If succeeds, the agent can call the Subscribe method of CAgent class, to subscribe any desired message group.

```cpp
void Subscribe(CString strGrpName, NOTIFY_CALLBACK callback, CString strTemplate="");
```

When an agent subscribes a message group, a callback function should be provided. Each time the Blackboard forward a message of this subscribed group to the agent, the given callback function will be called, giving the agent the chance to process the message. Agent should handle the message in the call back function. The following is the prototype of the call back function:

```cpp
typedef void (CAgent::* NOTIFY_CALLBACK) (CString & strMsg, CString& strCtnName, CString& strAgtName, CString& strReplyTag);
```

Step 7: Publish Message when Necessary

If the agent wants to publish a message to a message group, it can simply call the Publish method.

```cpp
void Publish(CString strGrpName, CString strContents, CString strReplyTag);
```

9.2 Smart Platform Application Wizard

Smart Platform Application Wizard is a very useful tool that generates the skeleton code of a new agent for MS VC++, which further ease the development of Smart Platform Agent. Figure 12
is a snap of the wizard. Basically the wizard asks the user to provide the basic information of the new agent, user’s preference of code generation and what kind of callback functions should be included in the new agent. Thus the new agent can be created with only some typing and mouse clicks.

One thing I’d like to mention is the format of the items in the Available callbacks list box, which is located on the right-bottom of the wizard. Firstly, those callbacks that start with a star are stream-oriented while others are message-oriented. Secondly, users can assign a message group to a callback (either message-oriented or stream oriented) and let the agent subscribe it when starts up, thus the callback is affixed by the name of the message group with an @ between them. For example, in Figure 12 the item “*StreamCallback@VideoStream” means StreamCallback is a stream-oriented callback method, which processes the stream data of VideoStream group, while “MessageCallback” means MessageCallback is a message-oriented callback and no message group is assigned to this callback, thus the subscribe method should be called with this callback as the second parameter before the callback can handle any messages.

![Figure 12](image)

Figure 12. A snap of the Smart Platform Application Wizard