AN ARCHITECTURE FOR PROVIDING ADVANCED TELECOMMUNICATION SERVICES

Yann Duponchel, Marcel Graf, Hong Linh Truong
IBM Research
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

Abstract We describe a novel service architecture that allows service providers to deploy and provision telecommunications services in an easy and efficient way. In contrast to today's Intelligent Network (IN) specification, the new architecture includes mechanisms for the automatic deployment, modification, and provisioning of services. It exploits the convergence towards IP as the universal network infrastructure to provide means for combining both Web and telephony services into more sophisticated and advanced ones.

Keywords: IN, Intelligent Network, IP Telephony, Service creation, Service provisioning, Communications services

1 INTRODUCTION

To accommodate the explosive growth of Internet traffic, network operators are building a high-capacity IP infrastructure, and there is no doubt that telephony will be just another application running on the same infrastructure. The cost advantage of managing and maintaining a single network infrastructure for all types of traffic is particularly important for newcomers, but less so for incumbent operators. Furthermore, using a single infrastructure will make it easier to combine telephony with Web-based services, thus creating new kinds of advanced and sophisticated services that have not been possible in the past or are difficult to implement on separate infrastructures. Such advanced services are in general thought to be the new income sources for which the service providers are desperately looking.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35522-1_37

H. R. van As (ed.), *Telecommunication Network Intelligence*

Before discussing how advanced services can best be created and deployed in an IP environment, let us review the basic architecture of an IP telephony service. Such an architecture is illustrated in Figure 1. Although it is based on ITU-T recommendation H.323 [1],[2], it can easily be shown that it also applies — with minor modifications — to other architectures that employ, for example, the IETF protocols SIP [3] or Megaco [4]. The most important difference between IP and legacy telephony architectures (e.g. PSTN, ISDN) is that the former does not require dedicated voice switches, because both signalling control information and voice signals use IP as communication means. Voice signals are sent directly between terminals, reusing the same IP router infrastructure as for the exchange of signalling information.

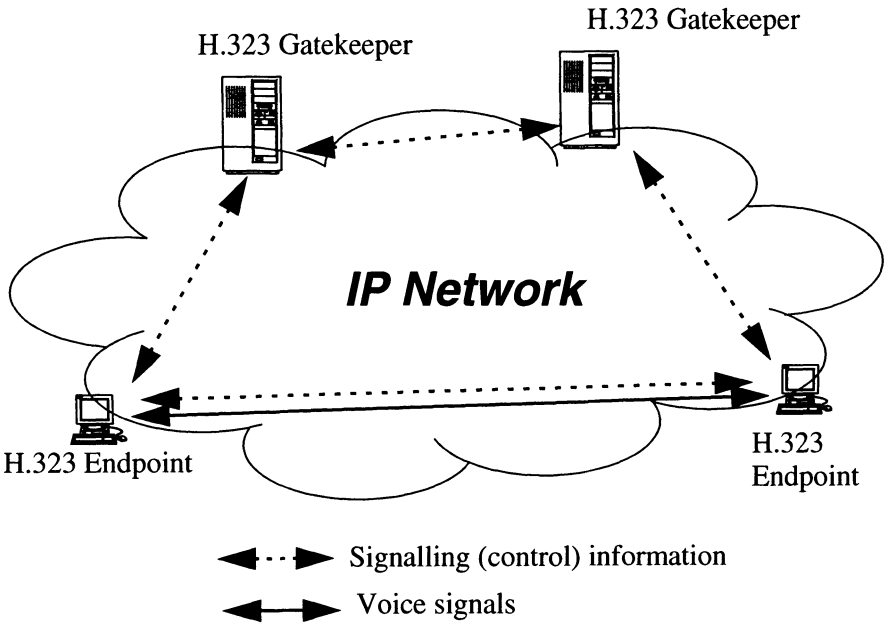

Figure 1: H.323-based IP telephony.

The gatekeepers shown in Figure 1 perform only control functions such as registration, access and admission control, address translation, etc. [2]. They have no voice switching capabilities and can therefore be implemented in software as applications servers.

Recommendation H.323 defines two methods to exchange signalling information, i.e. the information for controlling telephone calls:

1. *Direct method:* The signalling information needed for the control of a telephone call is exchanged directly between the endpoints involved. The endpoints may, however, need the assistance of a gatekeeper (or some other servers) for functions such as address resolution, access control, bandwidth management, etc.
2. *Gatekeeper-routed method*: In this case the terminals "talk" only to their gatekeeper, which now has control over the telephone call. The signalling information is exchanged between the terminals and their gatekeepers. The gatekeeper performs almost the same control functions as the legacy telephony switches, except that they do not switch voice signals.

In the context of service creation it is reasonable to foresee that such supplementary services as call forwarding or call transfer will be implemented directly in the endpoints rather than in the gatekeepers by exploiting the intelligence of the endpoints and the end-to-end capability of the direct method mentioned above. However the gatekeeper-routed method remains of interest to the service providers, not only because it gives them a control point for the telephone calls, but also because it enables them to offer more sophisticated services, e.g. by combining telephony with other Web-based information services. An example of such a combination is the stock alert service: the service subscriber is alerted, i.e. receives a phone call on any of his telephone sets (home, office, cellular, etc.), if the price of certain stocks crosses certain limits.

An incumbent telecommunications service provider, which already has a service creation infrastructure based on the Intelligent Network (IN) architecture [5], would certainly look for possibilities to reuse its existing IN infrastructure. Such a possibility is shown in Figure 2. The gatekeepers are considered Service Switching Points (SSPs), which means they intercept telephone calls and recognize when it is necessary to contact a Service Control Point (SCP) for further instructions on how to proceed with a call. The only difference between a gatekeeper and an authentic SSP is that the gatekeeper performs only call control, whereas a conventional SSP performs both call control and voice switching.

Although the architecture shown in Figure 2 allows incumbent telecommunications service providers to reuse their expensive SCP-based service creation infrastructure to extend the offering of traditional supplementary services to IP telephony users, it limits the range of offerable supplementary services to the capabilities of current SCPs and SSPs.

Another disadvantage of reusing the IN architecture is its lack of operation and management standardization: The IN architecture standardizes only the communication protocols between the SCPs and SSPs; it does not specify the mechanisms needed to "put" the service logics into the SCPs, to manage

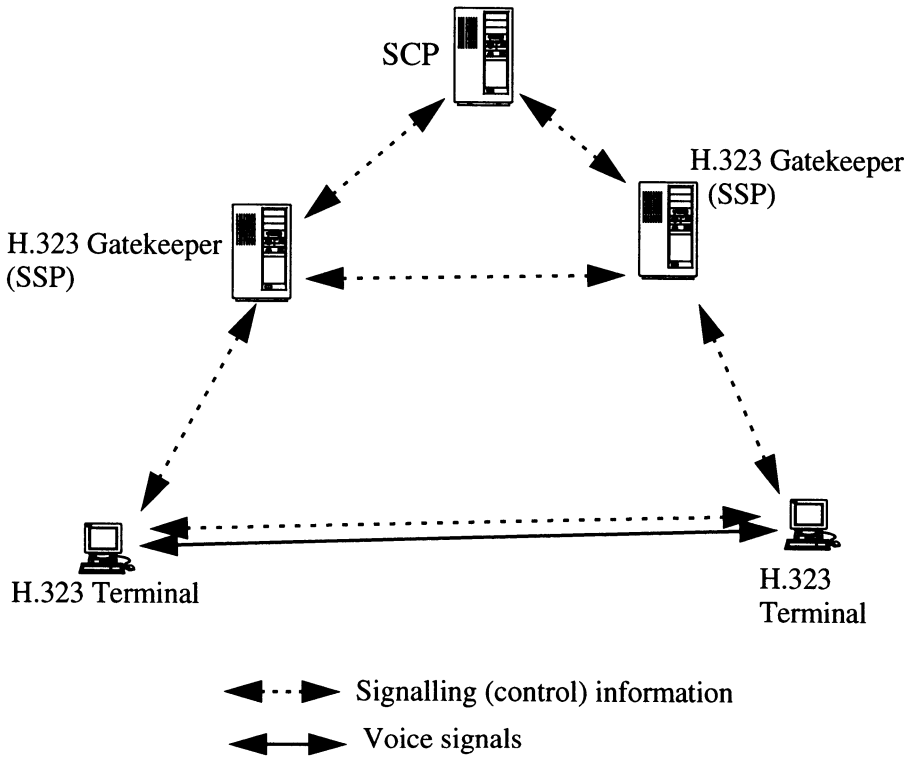

Figure 2: Applying IN architecture to IP telephony.

and customize them, and to set the required service triggers in the SSPs (e.g. recognizing the access prefix of a service). This lack of specification leads to incompatible and vendor-specific SCP/SSP platforms. Typically, a rapid deployment of services is only possible within a single vendor environment. Furthermore, it is not possible to port a service logic written for a specific platform to another, thus making it almost impossible to deploy and manage a service across multiple vendors' platforms.

In this paper we present a novel architecture that allows service providers to deploy and provision advanced telecommunications services in an easy and efficient way. In contrast to the IN architecture described above, the new architecture includes mechanisms for automatic deployment, modifications and provisioning of services. It provides to the service providers an easy and efficient means for injecting new service logics into the network and modify-

ing them. It also allows service subscribers to directly customize their service profiles. Besides being very resistant to failure of single components, the architecture is also scalable in terms of number of users, calls and services.

We have organized the paper as follows. In the next section we discuss some service concepts that are important for understanding our architecture. The new service architecture is then presented in Section 3, with a detailed description of the components and their interactions. Section 4 shows how a concrete IP telephony service based on the ITU-T Recommendation H.323 could be integrated into the architecture.

2 CORE AND ADVANCED SERVICES

In our architecture we differentiate between what we call *core* and *advanced* services. Core services are services used as common building blocks to create advanced services. Examples of core services are

- telephony service,
- e-mail,
- short message service (SMS)[1],
- instant messaging (IM), and
- presence.

Presence and instant messaging [6] is a new mode of communication that has recently become very popular in the Internet. Presence is a service that allows user A to declare its interest in the presence states of another user B, e.g. whether user B is currently connected to the network and, if so, whether user B is actively using his terminal, etc. This service is typically implemented by an application running on the terminal of user B, which publishes presence information about user B; the service delivers notifications to all users subscribed to observing each time the presence state of user B changes. Knowing the state of user B, user A may then start an instant messaging session with user B in which both users exchange short instant messages. An instant messaging session is very similar to a telephone call in the sense that it is synchronous (instant), i.e. messages sent by a user are delivered almost immediately to the peer user.

Although both presence and instant messaging are currently offered in the Internet as a combined service (e.g. AOL IM, Yahoo! Messenger) we regard them in our architecture as two separate core services that can be used independently to create new advanced services.

---

1. In the GSM mobile network, this service allows a cellular user to send a short text message to another cellular user.

An interesting observation is that most of today's telephony systems already collect presence information about their users, but do not offer it as an explicit service to their users. The PSTN, for example, monitors the status of a telephone line of a subscriber and determines whether it is busy or free. The fact of a telephone line going from a busy state to a not-busy state is a piece of presence information. Some systems make use of that information to offer so-called "call completion" supplementary services [7], in which a calling user for example can request the network to monitor a busy called user and connect him to that user as soon as he becomes not busy. Another example is the mobile telephone network, which foresees a procedure for mobile phones to register with the network before they can make and receive calls. The network keeps track of the registration status. The fact that a mobile telephone is registered is again a piece of presence information. It is interesting because it indicates that there is a high likelihood that the user is able to accept a call. A similar registration procedure also exits in IP telephony.

In our architecture we propose that all these different pieces of presence information be integrated into a single service available to all other ones. Based on such a converged presence service, the presence information is no longer restricted to a specific application: a user is no longer *available with instant messaging* or *available with telephony* but rather *available at a certain terminal.* In this way we can exploit the multiservice capability of modern communications terminals, e.g. a mobile phone that is capable of receiving telephone calls, SMS messages, and in the near future also instant messages.

Regarding the telephony core service mentioned above, it should be noted that in our architecture we do not differentiate between the different types of network technology that can offer telephony services: i.e., there is only one telephony core service, which includes the PSTN, ISDN, cellular networks, IP telephony networks, etc.

As mentioned, taking the core services as common building blocks, one can create advanced services by

- either modifying the way a core service is invoked, e.g. call forwarding, call completion. In the legacy telephony world such advanced services are called *"supplementary services"*;
- or combining several core services to provide more sophisticated ones. We call such advanced services *"hybrid services"*.

Of course an advanced service may be both supplementary and hybrid.

Our notion of hybrid services is different from the one defined in [8], in which it is defined as "services which span many network technologies, such as the public switched telephony network (PSTN), cellular networks, and networks based on IP". According to this definition, the telephony core service we mentioned above would already be a hybrid one.

3 THE NEW SERVICE ARCHITECTURE

The main aim of our architecture is to define an environment that will allow a rapid and efficient creation and deployment of advanced services.

In designing the service architecture emphasis was placed on the following key aspects. First, it should provide the service provider with an easy and efficient means to deploy new service logics into the network and to modify them. Second, it should allow service subscribers to customize their profiles themselves (e.g. via the Web and not with the help of a call center agent) and have the changes propagated rapidly into the network. Third, it should be applicable to the creation of hybrid advanced services as defined in Section 2, and not restricted to supplementary services. And last but not least, it should be scalable to large numbers of subscribers and services, and resistant to components failures.

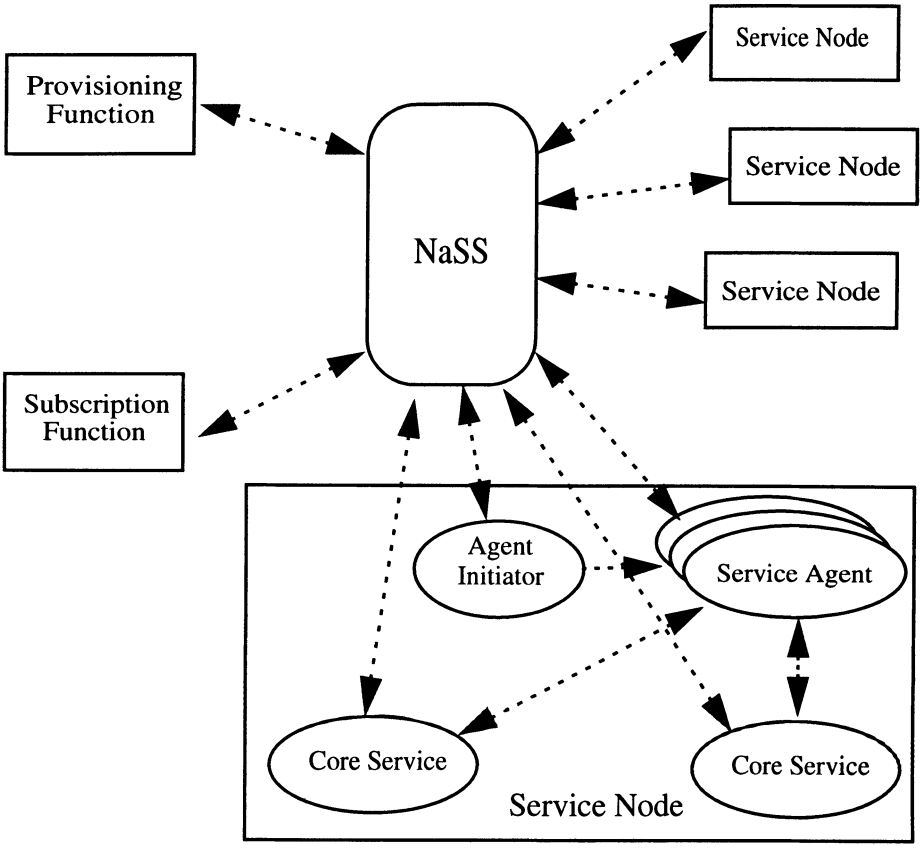

Figure 3: New service execution architecture.

The proposed service architecture is shown in Figure 3. It consists of the following components:

- Notification and Synchronization Service (NaSS),
- Provisioning Function,
- Subscription Function,
- Core Services,
- Service Agents,
- Agent Initiators, and
- Service Nodes.

We will first give a short overview on the functions of these components and their interactions. The follow-on subsections will then contain a more detailed description.

3.1 OVERVIEW

Typically there are four phases during the life cycle of a service [9]: creation, provision, subscription, and utilization.

During the creation phase, first the service is specified and designed, usually derived from a textual description, then the required service logics are developed, coded, verified and tested. Various formal languages such as SDL (Specification and Description Language) or LOTOS [10] can be of help during the specification and design steps. To simplify the coding tasks visual tools combined with reusable components such as the IN's concept of service-independent blocks (SIBs) or the object-oriented Java Beans can be used. The creation phase is, however, beyond the scope of our architecture, and for the remaining part of this paper, we assumed the existence of the service logics. Such service logics are named in our architecture *Service Agents*.

Having the service agents, the service provider now needs to distribute them to the *Service Nodes*, in which they will be loaded and executed during the service utilization phase. To be scalable our architecture supports the existence of multiple service nodes, which may or may not have the same functionality.

The *Provisioning Function* shown in Figure 3 helps the service provider to inject the service agents into the appropriate service nodes using the communication mechanisms of the so-called *Notification and Synchronization Service (NaSS)*. As will be described in more details in Section 3.2, the NaSS provides to all other components of the architecture an interprocess communication mechanism that is based on an anonymous publication and subscription paradigm. That means, a publisher does not need to know who will

receive his notification, he just publishes it under a certain name; the NaSS will then deliver the notification to the components that have subscribed to that name.

In our example, with the help of the provisioning function the service provider publishes the service agents into the NaSS, without the need for knowing anything about the characteristics of the available service nodes. The NaSS will then deliver them to the service nodes that have subscribed for those service agents. This procedure will be described in more detail in Section 3.6.

Similar to the provisioning function, the *Subscription Function* in Figure 3 allows a service user to subscribe and customize his service profile himself (e.g. using the Web) and publishes the resulting data into the NaSS. The NaSS will then deliver those data to the service agents that have subscribed for those data.

As mentioned, service agents execute the logic that is required to implement a certain advanced service. For this purpose they typically wait for an event coming from a *Core Service*, at which they then interact with that core service and/or with additional ones. For example, a service agent implementing the supplementary service "Call Forwarding on No Reply" (CFNR) [11] only becomes active if there is an incoming call to the subscriber. At this event (which is generated by the telephony core service) the service agent starts the no-reply timer and watches the state of the call. It does nothing if the call is answered, otherwise at the expiration of the no-reply timer, it disconnects the ringing connection and redirects the call towards the diverted-to number.

To make the service node scalable in large numbers of active service agents, we define in our architecture the concept of an *Agent Initiator*, which watches the events on behalf of the service agents. At the occurrence of an event the agent initiator will load into memory those agents that are interested in the event and start them. It is also the responsibility of the agent initiator to remove an agent from the node's memory after that agent has done its job. The concept of the agent initiator is explained in more detail in Section 3.4.

3.2 NOTIFICATION AND SYNCHRONIZATION SERVICE (NASS)

The heart of our architecture is the *Notification and Synchronization Service* (NaSS), which provides an interprocess communication mechanism to all the other components. As illustrated in Figure 4 there are two kinds of NaSS users: (1) NaSS *Publishers*, which send notifications to the NaSS, and (2) NaSS *Subscribers*, which receive notifications from the NaSS. The main

function of the NaSS is to deliver the notifications and attached objects sent by publishers to the corresponding subscribers. The association between publishers and subscribers is performed by means of a *Name*.

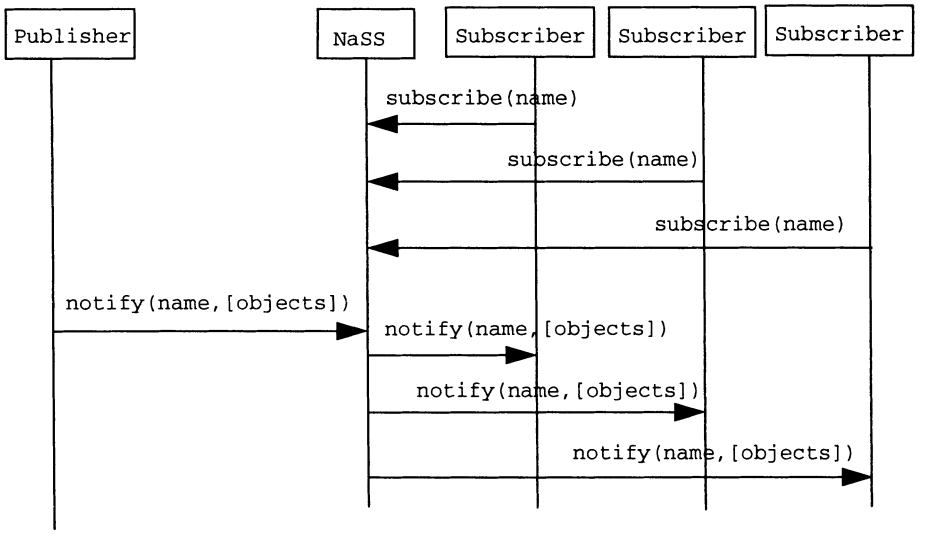

Figure 4: Notification and Synchronization Service (NaSS).

NaSS names have a hierarchical structure. This enables a subscriber to use wildcards to refer to an entire subtree of the hierarchy and thus to subscribe to several names with a single subscription.

Communication via the NaSS is *anonymous*, because to receive a notification sent by a certain publisher, a subscriber needs only know the name under which the publisher sends the notification (and not the publisher itself). The publisher also does not know who will receive its notification. Both publisher and subscriber do not even have to know whether the peer they want to communicate exists yet.

If there are no subscribers at the time of publication, the notification will be remembered by the NaSS for later delivery to new subscribers. Several publishers may publish notifications under the same name. All of them are delivered to the subscribers, if any. However, the NaSS remembers only the last notification, i.e. a new subscriber will receive only the last stored notification.

The above property and the anonymous communication are particularly important for the subscription phase of a service, in which a service subscriber modifies its profile and publishes the new data into the system using the Subscription Function (see Figure 3). This data is needed by a service

agent activated, for example, when the subscriber is involved in a telephone call. The anonymous communication property allows the subscription function to inject the data into the system without the need to know which agents will get it and on which nodes they will run. The memory property (remembering the last notification) allows the service agents to always get the most up-to-date data.

The NaSS service is both synchronous and asynchronous. It is synchronous because it guarantees the latency with which notifications are delivered to subscribers. At at the same time it is an asynchronous service because it remembers the last notification and delivers it to a new subscriber.

In general, all components communicate with each other using the NaSS publication and subscription communication mechanism, without the need to know the identity and location of the peer component. The NaSS is also used for local communication. This permits the network operator to place any component anywhere in the network without the need for complicated configuration work.

The NaSS concept of anonymous and asynchronous communication described above is actually not a novel concept. It has its root already back in the 1980's in the field of parallel computing [12] and has recently become again an interesting research area in the field of Java-based distributed computing and mobile agents [13],[14]. New is however our proposal for applying this concept to the field of telecommunication service deployment and execution and its extension from an asynchronous messaging system to a synchronous one to fulfill the real-time requirement of telecommunications services.

3.3 CORE SERVICES

As mentioned in Section 2, *Core Services* are the building blocks for creating new and sophisticated advanced services. In our architecture there are two ways the other components can access the functionality of a core service:

1. Through the NaSS
   In this case the core service can be accessed by all components, independent of their location.
2. Directly
   In this case the core service is primarily accessible by components residing on the same service node (the concept of a service node is defined in Section 3.5). It may not be available to components residing on other service nodes.

The direct access type has the advantage of better performance in those cases where interaction with the core service requires an intensive exchange of notifications. If the component accessing the core service resided on a different service node, the resulting high network communication overhead would impose a significant performance penalty. An example of a core service that would need direct access is a call-control service using the Java Telephony API [15], which has a very detailed call model. In JTAPI, a telephone call is represented as a set of finite state machines that undergo state transitions. Each state transition is delivered using the "Event Object" pattern [16] to the service logic. The number of events generated by a call is very high when compared, for example, to the IN call model. Thus, a component that wants to control a call offered by a JTAPI interface needs be colocated with that JTAPI.

Providing access to a core service does not mean that a platform conforming to our architecture also has to implement that core service. If the core service is already provided by another system, then of course the platform will not implement that core service again but only provides the means to access and control it. For example, in the case of the telephony core service the platform will not implement the telephony service again, but merely provides the means for controlling telephone calls to its components. The telephony service itself is implemented elsewhere, e.g. by the PSTN, by a cellular system, or by IP telephony system.

3.4 AGENT INITIATORS AND SERVICE AGENTS

The main function of a *Service Agent* is to execute the logic that is specific to a certain advanced service. Service agents are designed and created by the service creator. In general several service agents are needed to implement one advanced service. The architecture does not assume any specific relation between a service agent and the phases of the service life cycle, i.e. during a certain service phase, multiple service agents may be executed in parallel, and an service agent may also cover multiple phases. The architecture also neglects the interactions that may arise between various service agents. It is the responsibility of the service creation to resolve the feature interaction problems [17] during the service specification and design phases.

Typically a service agent is in passive mode most of the time. It becomes active only at the occurrence of a certain event (e.g. an incoming call), executes the required actions, and returns to the passive mode. Because there may be a large number of service agents that need to listen for their triggering event, they all would have to be loaded into memory all the time, independent of whether they are passive or active.

To be scalable in a large number of service agents, we introduce in our architecture the concept of an *Agent Initiator* that listens for the relevant events on behalf of the service agents. At the occurrence of an event, the agent initiator loads the corresponding service agents into memory and initiates them. Once initiated, a service agent runs on its own, i.e. it interacts directly with the other components, without any help from the Agent Initiator. In addition, the agent initiator is responsible for unloading the service agents from memory when they have done their job and return to the passive mode.

The agent initiator needs the following information to be able to load and initiate a service agent:

1. The *events* at which the agent initiator has to load and initiate the agent;
2. the *code* to be loaded (this is the code that implements the logic to be executed by the agent once it becomes active);
3. the *data objects* needed by the agent to do its job, and
4. the *core services* the service agent needs to access to do its job, e.g. telephony core service. This information is important for core services that are accessible only directly.

Thus when creating the service agents the service creator not only needs to create the code of the agents, but also provide a so-called *Service Agent Descriptor* that describes these four pieces of information to the agent initiator. As in our architecture all information exchange is performed via the NaSS, a service agent descriptor contains the NaSS names under which the agent initiator has to subscribe to obtain the information required:

- Event names: the NaSS names of the events that will trigger the initiation of the agent;
- Code names: the NaSS names under which the agent code can be retrieved;
- Data names: the NaSS names under which the data objects needed by the agent can be retrieved, and
- Core services names: the NaSS names of the core services needed by the agent.

During the provisioning phase the service agents' codes and their descriptors are made available to the agent initiators by publishing them in the NaSS. As a consequence, at service node start-up time, the agent initiator subscribes to the NaSS for all service agent descriptors in order to get the service agents made available by the service provider. Section 3.6 provides more details on the use of NaSS during the provisioning phase of a service.

Upon receiving a service agent descriptor, the agent initiator checks the core services descriptor to find out whether it is able to support that service agent (it is assumed that the agent initiator knows the capabilities of the ser-

vice node it resides on). If not, the published service agent will be ignored. Otherwise, the agent initiator subscribes to the NaSS or a core service interface (e.g. in the case of JTAPI) for the events specified in the service agent's event descriptor.

Upon occurrence of one of the events described in the service agent's event descriptor, the agent initiator

- subscribes to the NaSS to get the agent code (if this has not already been done);
- subscribes to the NaSS to get the data objects specified by the service agent's data descriptor, and
- starts the service agent's code with the data objects it gets from the NaSS.

As the NaSS functionality is available to all components of the architecture, the service agents can also access the NaSS directly to get the information they need to do their jobs rather than getting it from the agent initiator. Our architecture provides both mechanisms, and it is up to the service creator to select the one that is most appropriate to his services.

Note that the agent initiator is agnostic about

- the number of service agents a certain advanced service may need. It is the responsibility of the service creator to define the number of service agents needed by a service. For example, if the service requires actions at multiple events, then the service creator may define multiple agents, one for each of the events. He may also create a single one which listens to all events (except the first one, which is delegated to the agent initiator);
- the number of subscribers involved by a service agent (i.e. the logic executed by a service agent may be specific for a single subscriber, but also may be valid for a group of subscribers), and
- the relationship between a service agent and the service life cycle.

3.5 SERVICE NODE

A service node is a functional grouping that contains at least an agent initiator and is therefore able to initiate service agents. Furthermore it provides to the components it hosts access to the NaSS for communicating and exchanging information with other components.

In general, there are multiple service nodes within a system implementing our architecture. To be scalable in terms of the number of users it can accommodate, the processing load generated in the entire system needs to be spread over several service nodes. There are two possible approaches regarding the division of the load: by function (task) or by load-generating request. In the

case of division by function a node specializes in performing only certain subtasks. To handle a request several service nodes need to communicate to coordinate the subtasks. In the case of division by load-generating request, a service node performs all subtasks for a request. As a request enters the system it is assigned to one service node.

The latter approach in general has some advantages over the former because (1) it has a lower communication overhead, (2) is more robust because the load of a failed service node can be taken over by the other service nodes, and (3) easier to manage when the system needs to be scaled up as another service node can be merely be added.

Our architecture allows the implementation of both approaches, even a combination of them. The concept of agent initiators and service agents, together with the NaSS communication mechanisms, permits the addition and removal of advanced services without the need for re-configuring and restarting the service nodes. New service agents are distributed automatically to the subscribed service nodes and started there without the need for complicated configuration work. It also allows the addition and removal of service nodes to distribute load over multiple machines, thus making the architecture scalable in terms of the numbers of users, calls and services that can be accommodated. It furthermore adds reliability to the system because a failing node can easily be replaced by another one.

3.6 PROVISIONING AND SUBSCRIPTION FUNCTIONS

Assuming the existence of the service agents coded as described in Section 3.4, the *provisioning function* shown in Figure 3 helps the operator inject the service agents into the network using the publishing mechanism of the NaSS (provision phase). For example, the operator can publish the service agent implementing the CFNR supplementary service [16] using the NaSS name `/serviceagent/descriptor/cfnr`. It is the responsibility of the service creator to choose the name such that it is unique in the system. With this naming scheme, the Agent Initiator would then have to subscribe to the name `/serviceagent/descriptor/*`. The wildcard `*` is used to refer to all notifications published using names that start with `/serviceagent/descriptor/`. In this way all service agent descriptors published via the provisioning function will be forwarded by the NaSS to the Agent Initiators.

Certainly, by giving a more complex structure to the NaSS names, specialized service nodes and agent initiators could be introduced into the network. In this case, these nodes would subscribe only to (and therefore get only) the agents of the services in which they were interested. This example

shows the strength of our architecture: service logics can easily be transferred to the appropriate network nodes, without the need to know the location and configuration of the various nodes explicitly.

This also applies to the subscription phase, in which a user can subscribe to a service and customize his service profile according to his needs. The *subscription function* in Figure 3, besides interfacing with the user, publishes the most up-to-date subscriber data into the system. In this way any components in the system, e.g. service agents, currently subscribed to those data always get the most recent one from the NaSS.

Although the subscription function is shown in Figure 3 as a component different from a service node, it can certainly be designed in a very similar way. This means that it will also contain an agent initiator for initiating service agents, which, in this case, will run during the subscription phase and not, as in the case of the service nodes, during the utilization phase. These "subscription" agents are also created by the creator and injected into the network using the provision function described above.

4 INTEGRATION OF H.323-BASED TELEPHONY SERVICE INTO THE ARCHITECTURE

In the preceding sections the inclusion of the telephony core service was handled in a very generic way, i.e. without specific assumptions on how that service is actually implemented. Whether it is provided by a legacy PSTN or an IP-based network was not important.

Figure 5 now shows how the architecture looks if the core telephony service is based on H.323. The gatekeeper-routed call model is selected because we want to provide the service agents running in the service nodes with the capability of controlling the H.323 calls. The H.323 gatekeeper functions, as specified in [1], are split into two parts:

1. The *H.323 signalling proxy*, which mainly handles the signalling functions of the Q.931 and H.245 protocols and provides the telephony call control services to the other components of the service node, and
2. the *H.323 RAS Server*, which covers the functions defined by the H.225.0 RAS protocol such as address translation, admission control, etc.

The separation of the gatekeeper functions into a signalling and a RAS part allows the implementation of a very efficient load distribution mechanism. All service nodes use the NaSS communication mechanism to publish

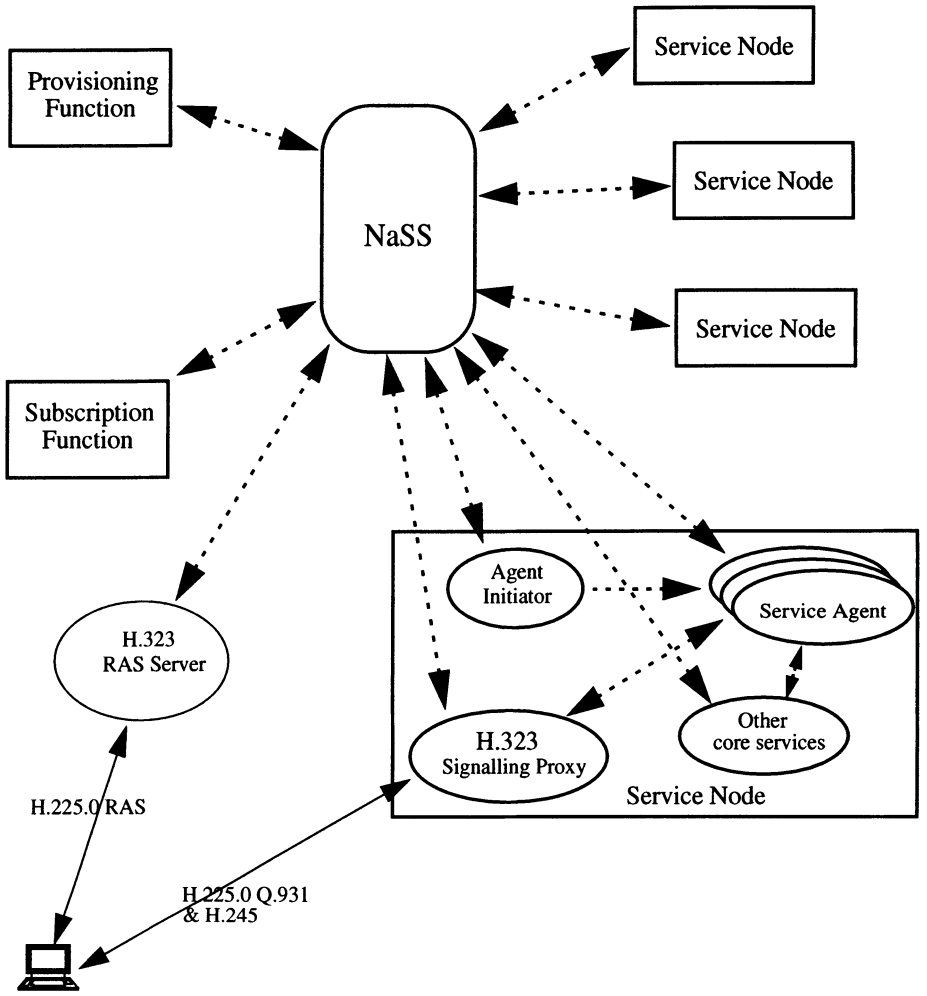

Figure 5: Integration of H.323-based telephony service into the architecture.

information about their availability and load situation. Because the communication via the NaSS is anonymous, they do not know the existence of the RAS server, and even do not know who will manage their load.

To get the load information of the service nodes, the RAS server responsible subscribes to NaSS with the name used by the service nodes to publish their load information. Based on the load information collected, the RAS

server is able to construct an overview of the actual load situation of the service nodes and can therefore distribute new calls to the most appropriate nodes. The entire scenario is illustrated in Figure 6, which starts with the RAS server subscribing to the load notifications sent by the service nodes. According to Recommendation H.323, an endpoint needs to send an *admission request (ARQ)* message to the gatekeeper (in our case to the RAS server) before it can set up a call. The RAS server will then request the endpoint to send its Q.931 SETUP to the most appropriate service node. The IP address of the selected service node is given back to the endpoint via the *admission confirm (ACF)* message.

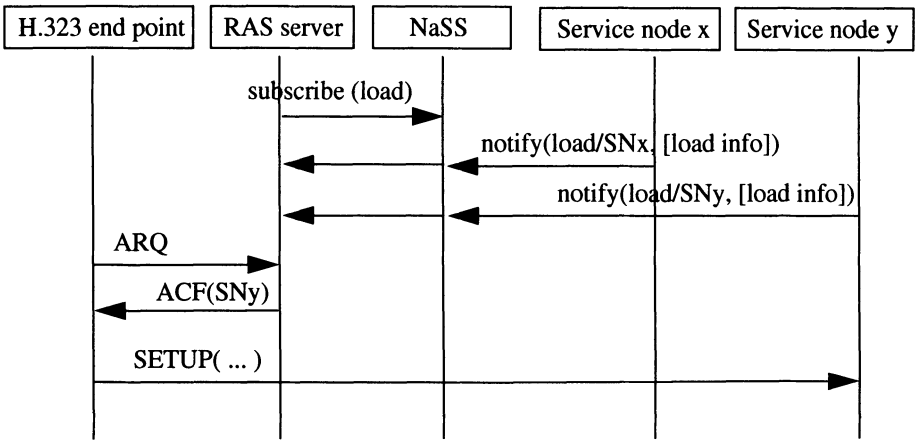

Figure 6: Load distribution.

The NaSS capability can furthermore be exploited to implement the H.323 address registration and translation. In H.323 an endpoint may be assigned an alias that is independent of its transport address. This is important for example for endpoints that access the Internet via a dial-up connection with dynamic IP address assignment. To be nevertheless always reachable under the same alias, the endpoint has to inform the gatekeeper about its current transport address using the so-called registration procedure.

Figure 7 illustrates how the H.323 address registration and translation can be implemented using the NaSS capability. The endpoint informs the gatekeeper (in our case the RAS server) about its current transport address by means of a *registration request (RRQ)* message. Upon receiving an RRQ from an endpoint, the RAS server publishes the received H.323 alias address and the corresponding transport address into the NaSS, thus making the translation immediately available to all components of the system. Any compo-

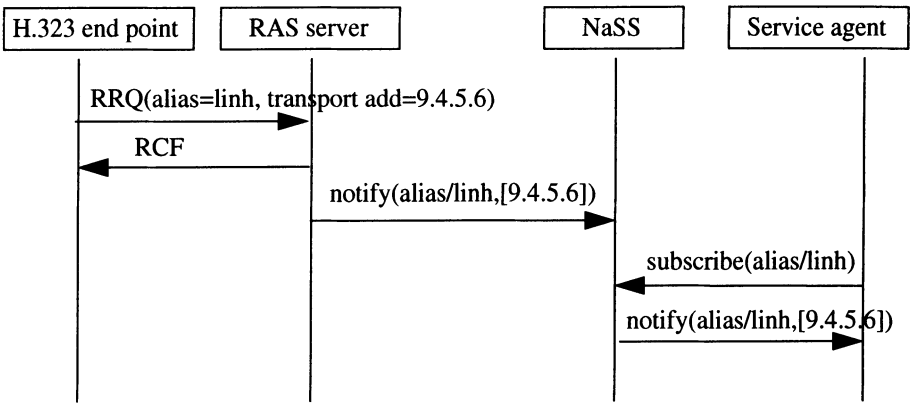

Figure 7: H.323 address registration and translation.

nent, e.g. a service agent that needs to set up a call to a certain endpoint, can retrieve the actual transport address of that endpoint simply by subscribing to the NaSS for the alias associated with that endpoint. To remove an association (e.g. in the case of an unregistration), the RAS server publishes an empty association, i.e. a notification that contains an H.323 alias but without a corresponding transport address. Note that all H.323 RAS specifics, e.g. time to live, etc., are implemented in the RAS server, not in the NaSS. The NaSS stores only the association between an alias and a transport address.

Although the architecture illustrated in Figure 5 was developed based on the ITU-T H.323 recommendation, it is obvious that it can also apply to the IETF SIP protocol by replacing the H.323 RAS server and signalling proxy by a SIP redirect and proxy servers, respectively.

5 CONCLUSION

We have described a novel architecture that allows the rapid and efficient creation and deployment of sophisticated advanced services. The core component of the architecture, the NaSS, provides the service creator with an easy and efficient means for injecting new service logics into the network and modifying them. It also allows service subscribers to customize their service profiles directly. The concept of service agents and agent initiators permits the addition or removal of services without the need to restart the service nodes. It also permits the addition and removal of service nodes to distribute load over multiple machines, thus making the architecture scalable in terms

of the number of users, calls and services it can accommodate. The dynamic task assignment adds reliability to the system because a failing service node can easily be replaced by another one.

Our future work consists of building a prototype to assess the correctness of the concepts we have presented in this paper. We are evaluating various implementations of the NaSS, in particular regarding the fulfillment of the requirements described in Section 3.2. A performance evaluation of the architecture using a typical hardware and software platform also remains to be done. The results of these activities will be communicated in future publications.

ACKNOWLEDGMENT

The authors are thankful to Lucas Heusler for many insightful discussions and an in-depth review of the paper. They are also grateful to Asser Tantawi, Linda Steinmuller, Pnina Vortmann, Samuel Kallner and Igal Golan for useful comments.

References

[1] ITU-T Recommendation H.323, "Packet-based Multimedia Communications System", Version 2, Oct. 1999

[2] J. Toga and J. Ott, "ITU-T Standardization Activities for Interactive Multimedia Communications on Packet-based Networks: H.323 and Related Recommendations", Computer Networks 31, Feb. 1999

[3] H. Schulzrinne and J. Rosenberg, "Internet Telephony: Architecture and Protocols - an IETF Perspective", Computer Networks 31, Feb. 1999

[4] F. Cuervo et al., "Megaco Protocol", draft-ietf-megaco-protocol-07.txt, Feb. 2000

[5] I. Faynberg, L.R. Gabuzda, M.P. Kaplan, and N.J. Shah, "The Intelligent Network Standards", McGraw Hill, 1997

[6] M. Day, J. Rosenberg, and H. Sugano, "A Model for Presence and Instant Messaging", RFC 2778, Feb. 2000

[7] ECMA Standard 185, "Call Completion Supplementary Services", Dec. 1992

[8] C. Gbaguidi, J.-P. Hubaux, G. Pacifici, and A.N. Tantawi, "Integration of Internet and Telecommunications: An Architecture for Hybrid Services", IEEE JSAC, Vol. 17, No. 9, Sept. 1999

[9] ETSI Technical Report ETR323, "Intelligent Network (IN): Service Life Cycle Reference Model for Services Supported by an IN", Dec. 1996

[10] K.J. Turner (Editor), "Using Formal Description Techniques - An Introduction to Estelle, LOTOS and SDL", John Wiley & Sons, 1992

[11] ECMA Standard 173, "Diversion Supplementary Services", June 1992
[12] S. Ahuja, N. Carriero, and D. Gelernter, "Linda and Friends", Computer, pp. 26-34, August 1986
[13] E. Freeman, S. Hupfer, K. Arnold, "JavaSpaces(TM) Principles, Patterns and Practice (The Jini(TM) Technogy Series)", Addison-Wesley, Reading, 1999
[14] T.J. Lehman, S.W. McLaughry, and P. Wyckoff, "TSpaces: The Next Wave", Hawaii Int'l Conf. on System Sciences (HICSS-32), Jan. 1999
[15] "JTAPI Introduction", http://java.sun.com/products/jtapi/
[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, Reading, 1995
[17] F.S. Dworak et al., "Feature Interaction Problem in Telecomunication Systems", Proc. of the 7th Int'l Conf. on Software Engineering for Telecommunication Switching Systems, pp. 59-62, July 1989, Bournemouth, UK.