Introducing Dynamic Distributed Coordination in Web Services for Next Generation Service Platforms

Muhammad Mukarram Bin Tariq, Toshiro Kawahara
DoCoMo Communication Laboratories USA, Inc.
{tariq, kawahara}@docomolabs-usa.com

Abstract

We present a technique called Dynamic Distributed Service Coordination Protocol (DDSCP) that enables dynamic and distributed coordination for composed services and applications in telecommunication networks. Individual service components are modeled as web services and DDSCP facilitates coordination among these components by dispatching executable processes to the service components that specify different steps that the service component must follow in response to (receipt of) specific messages and events. The collective and concurrent execution of these processes at different service components achieves overall goals of the service. The planning and creation of these processes is not our focus in this paper. We describe the structure and processing of different messages DDSCP, and describe how this protocol can work.

Our model has several advantages over the existing service platforms for 3rd Generation Mobile Networks, such as Parlay/OSA, and the web-service composition models. These advantages include introduction of flexibility among network components at finest level, ease of creation of highly customized services, easy integration with foreign components, reduced application complexity, increased reuse of application components, and possibility of increased user participation in managing her services, and thus reducing load on the network.

1. Introduction and Overview

The success of future mobile telecommunications systems depends on their ability to provide a steady stream of useful and highly customizable application. Two main challenges in providing highly customized services in the networks are: 1) how to create new services, and 2) how to manage these in a scalable manner. In this paper, we present a new framework and a associated protocol that allows highly customizable services. We believe that using this protocol in conjunction with a loosely coupled model, such as that of web-services, we can largely overcome these challenges. It also extends the web-services architecture beyond its current model.

Loose coupling and programmability are widely accepted solutions to the first of the two challenges we mentioned above. In a loosely coupled network, the network components are largely independent of each other; they can work and evolve independently without having much effect on each other. This eases the introduction of new functionalities and allows fast evolution of the network, which is the primary substrate for new services. Programmability refers to ability to change the behavior of the network and its components after initial deployment. This allows that the network to meet the requirements of future services and any problems that are discovered in course of time are easily fixed. Programmability also refers to be able to write applications that benefit from the baseline service components.

The second challenge, scalable management, has several aspects. In this paper, however, we are primarily concerned with manageability of customized services, or applications in a telecommunications network, and within that, how the services are created and provisioned, how they are coordinated.

Present telecom networks are tightly coupled, offer low programmability and manage applications in way that is a non-scalable, certainly in the face of expected wave of highly personalized services. Parlay [4], and its 3GPP adoption, Open Service Access (OSA)[3], which represent the state of the art in service platforms for telecommunication networks, provide an API that, in principle, allows programmability and creation of new applications. However, the “wrap-around” nature of this API leaves the internal rigidity of the network intact. Even with grouping the network functionality in to service capability functions, the network can, at best, be seen as one large component, instead a collection of several small independent and loosely coupled components. MExE [14] and USAT [15] allow some degree of involvement of user devices in the process of service creation and execution, but much of the burden of service provisioning remains on the network. The network is responsible for providing services, managing user profiles, and ensuring that the users get a service that is in accordance with their profiles.

In this paper, we present an alternative model that tackles the shortcomings of the existing service platforms. For loose coupling, our proposal is to view the network as a collection of several small independent components, based on web-services model, wherever practical. We refer to these components as service components. Programmability is available in form of customized applications that consist of sets of dynamically deployable customized processes. These processes, once deployed, not only *customize* the behavior of the web-service on which they are installed, essentially creating a custom agent. These processes also enable distributed coordination between service components, allowing them to work concurrently towards the goal of the application, just like in multi-agent systems.

Moreover, in our model, the users can store these distributed applications as part of their portfolio, possibly on their device, or create them on the fly, such as a part of service composition, and trigger their deployment on demand. This frees up the network of cumbersome management responsibility.

Recognizing the fact, that network is often a shared infrastructure among millions of users, the framework has some built-in security, and safety features. These features allow deployment of only authorized processes, and provide hooks through which the service components can ensure that the processes for one user do not interfere with services for others.

In this paper, we do not address the exact mechanism through which these applications and their constituent processes are created. It could be done by application developers, or by automated agents, and could be based on the service composition methods. Our focus in this paper is a framework that defines the structure of these processes, and how they are distributed and how the service components process them to achieve dynamic and distributed coordination leading to achievement of overall goals of a service.

The core of this framework is a protocol, which we refer as the Dynamic Distributed Service Coordination Protocol or DDSCP. Its primary responsibility is distribution of the processes. The processes themselves are expressed as workflows, specifically in BPEL4WS [8]; we do propose some changes and new activities to accommodate the dynamic deployment requirement.

The basic model is that a service initiator dispatches a DDSCP *InstallProcess* message to the service components. This message contains a process along with a set of parameters that define the subset of messages and the events that a process is interested in processing. The *InstallProcess* message also contains information related to integrity of these processes, and some preprocessing activities, that allow a receiving service component to customize the process further before execution. The execution of the process can trigger dispatch of processes to other service components.

Our model allows several service initiators to "install" their own customized versions of processes for the *same* service components, so that the service component behaves differently for each of these initiators, however, the local policy and safety rules of the component can restrict the privilege of installing process and the types of messages that the these processes can handle.

The structure of rest of this paper is as follows. In section 2, we present related work. Section 3 is dedicated to detailed description of DDSCP. In section 4 we present the overall service platform model, that describes the modified web-service structure to allow use of DDSCP, and puts use of DDSCP in context with other components of the service platform. This section also describes advantages of overall model and different features of DDSCP. In section 5, we present examples that illustrate the working of the DDSCP. In section 6, we discuss various aspects and issues related to the new service platform and the service coordination protocol. We close the paper with a word on conclusion and future work in the section 7.

2. Related Work

Our work relates to a number of areas. These include the service platforms for existing networks, customizable web-services, personalization techniques in the Internet services, dynamic distributed coordination of activities with workflows, programmable networks, such as Active Networks, and more fundamentally, the object oriented concurrent programming techniques.

Coordination is an important step in any component-based model. Coordination can be centralized or distributed. In centralized coordination, a single entity communicates with all other components, and maintains the state of the overall composition. In distributed coordination, there are several coordination points that are responsible to coordinated different aspects of the overall composition. Distributed coordination, while relatively hard to conceptualize and manage, has several advantages over the centralized coordination approach, including scalability, reuse, and robustness. Rainmaker, by Paul et al. [10][11], is a distributed workflow infrastructure where a source can dynamically assign tasks to performers, achieving a

distributed coordination model. This model serves as basis for distributed coordination in our work as well, where we can roughly characterize the service originator or the user as the source, and different service components as performers. Our protocol provides a realization of this model that is suitable for web services. In our model, a service component can be working for a number users, or service originators, so the security and safety are a big concern. We provide mechanisms related to security and safety so that the processes of one user may not interfere with those of others, and that the service components only accept processes from trusted entities.

Active Networks (AN) [1][2] held immense promise to provide fine grain customized behavior from network nodes through programmability. However, in our opinion, one of their disadvantages, which also impeded their wide adoption, was the need for the messages and packets to be aware of active networking and carry a reference to the process that must execute on the message. This requirement, although allowed fine control as to which process is executed, was not practical, and a considerable hindrance in the evolution of AN. Active Packets and Plug-in extensions for AN tried to isolate the effect of evolution, but the basic model of packet carrying the reference remained unchanged. Recently, Song et al. [12] hinted on this limitation of AN, and extend AN model to support customized processing without requiring packets to carry a reference. We also avoid the requirement of unsuspecting messages carrying the reference to the process, for our framework.

Finally, the notion of independent and customizable web-services components working towards a goal is fundamentally similar to object oriented concurrent programming and concurrency in open systems and Agents paradigm. Actors [7] allow reconfigurability and inherent concurrency. Actors can send messages to themselves and to other actors, create new actors, and specify replacement behaviors. The process deployment on service components using DDSCP is equivalent to the creation of new actors (customized web-service) and specifying a replacement behavior. Just like actors, web-services have address binding and can send and receive messages to each other. Hence, we can use the Actors to reason about web-services and dynamic behavioral updates in web-services, although we do present any formal analysis in this paper.

3. DDSCP Messages and Processing

DDSCP is a simple protocol whose primary purpose is to distribute the processes to the service components. The protocol includes mechanisms through which the receiving service components can verify the integrity of the processes. It also provides certain parameters that the processes can use to correlate their activities.

3.1. DDSCP Messages

InstallProcess message is of primary importance in DDSCP. This message contains an executable process in its payload, along with process properties, correlation parameters and security related information. Figure 1, shows a simplified XML representation of this message.

First part of the message is the *InstallProcessHeader* element. This element includes information about the process correlation and security. One of the concerns of a service component receiving a foreign process is that the process is coming from a trusted entity[1]. The message header includes information about a primary authority that has authorized the process in the payload of the message to be executed as part of a set of processes identified by the *ProcessGroupIdentifier*. A service component only executes the process if it trusts the primary authority.

```
<InstallProcess
   <InstallProcessHeader>
      <ProcessAuthority>
         <PrimaryAuthority/>
         <DelegatedAuthority/>
         <ProcessInvokeAuthority/>
      </ProcessAuthority>
      <ProcessGroupIdentifier/>
   </InstallProcessHeader >
   <ProcessProperties>
      <ProcessIdentifier/>
      <CorrelationParameters/>
      <ExpectedElements/>
   </ProcessProperties>
   <PreProcessActivities/>
   <PreProcessActivitiesData/>
   <process>
      <activity 1>
      <activity 2>
      <dispatchProcess>
          <Destination Information>
          <PreDispatchActivity>
          <Process>
      </dispatchProcess>
      …
      <activity N>
   </process>
</InstallProcess>
```

Figure 1. The DDSCP InstallProcess Message

The header also includes a list of invocation authority identifiers (public keys) and a process identifier tuples. The tuples bind the entity with the process identifier that the primary authority allows it to dispatch or invoke at another service component. To enhance process reuse, it may be necessary to defer the invocation authorities binding. For this purpose, the primary authority assigns an intermediate or delegate

[1] There are other, arguably more important concerns, such as, safety of the process, but they are orthogonal to the issue of transport of the process, and a number of schemes already exist that can verify certain safety properties in a given piece of code.

authority, bound to the ProcessGroupIdentifier. This authority can alter the binding of invocation authority and process identifier dynamically, and include as a signed separate element. The primary authority signs the entire InstallProcessHeader element to ensure its integrity.

Next in the message is the element containing process properties. These properties include an identifier for the current process (processIdentifier) bound, using public key of the primary or delegated authority, to the hash of the process included in the payload, the processGroupIdentifier included in the message header, and the identity of the receiving service component. The primary or delegate authority also binds the names of elements that the receiving service component must expect in the message. This is to prevent malicious entities from stripping off parts of the message and causing unintended behavior.

An important part is the correlation parameters. Unlike BPEL4WS, we require declaration of these parameters ahead of installation. These parameters define the subset of messages that the process is looking to handle, or for which the process wants to provide customized behavior. Depending on the policy of the service component, the process may be required to declare such parameters for both incoming and outgoing messages.

Other properties in the process properties element define whether and where the service component should report on completion of execution of the process or if any errors occur during the execution of the process. The process properties element must be signed by one of the authorities identified for this process in the header.

The optional PreProcessActivities element includes a set of activities that the receiving service component must perform before executing the subsequent process. The element may include activities in same format as the main process, and may additionally include a transform that operates on the main process to prepare it for execution. The reason for inclusion of this capability is that depending on the conditions under which the sending entity dispatches the InstallProcess message, the author of the application may want the process of the receiving service component to be modified. In simplest cases, the transform may initiate the process differently, and it more complicated scenarios, the transform may make radical changes in the process, such as for context aware adaptation. However, the later is not recommended due to safety concerns. In order to protect against malicious intermediaries, the PreProcessActivities element is bound to the process it is intended for, by using digital signature of the primary or delegate authority over the PreProcessActivities element and the hash of the process. The "conditions" that make the process modification necessary are of course dynamic in nature and cannot be pre-signed by an authority. These are included in the message as a separate element called the PreProcessActivitiesData element by the dispatching entity. These conditions serve as input to the PreProcessActivities.

The last element in the message is the process that the service component must install and execute. More precisely, service component executes the output of the transform, if such a transform is present. The element is signed by the primary or delegated authority, and the service component must verify the signatures prior to applying the transform.

Other messages in DDSCP include IntallProcessAccept, InstallProcessReject and messages for interacting with the checkpoint service (that we will describe shortly). The IntallProcessAccept message indicates to the sender that the receiving service component has accepted the process that was earlier sent in an InstallProcess message. The InstallProcessReject message indicates that the receiving service component did not accept the process. The message body optionally includes the reasons. Apart from these, there are messages to set and retrieve check-pointed information; we will describe this feature shortly.

As indicated before, we are presently using BPEL4WS as the language to specify the process; however, we have defined additional activities that facilitate the distributed coordination. Description of these activities follows next.

3.2. New Activities for DDSCP

We define three new activities within the BPEL4WS framework to facilitate different aspects of dynamic distributed control. First of these is the *DispatchProcess* activity. This activity instructs the processing service component to dispatch a process to another entity. The process to be dispatched is included inside the DispatchProcess activity element. The DispatchProcess activity element also includes the destination of the process to be dispatched and optional PreDispatchActivities element that contains a set of activities that the sending service component must perform before dispatching the process to the destination. The PreDispatchActivities element has the same format as the PreProcessActivities that we presented in previous subsection, and its purpose is to is to generate the PreProcessActivitiesData element

that the service component receiving the process will use in its pre-process activities. Explanation of processing of DDSCP messages, given in the next sub-section, will clarify this further.

Second of the new activities is *AbortProcess* activity. This activity includes a process identifier, along with optional correlation parameters needed to identify the correct instance of the process that should be aborted. The service component executing the process containing this activity verifies whether the identified process exists locally, and whether the invocation authority of the current process is an authority on the process to be aborted. If the service component successfully verifies both of these conditions, it immediately aborts the process.

Last of the three new activities is CheckPoint activity. The activity identifies a checkpoint server and a set of elements whose value the service components should send to the checkpoint server. This multipurpose activity may be used for inspection and safety.

3.3. Processing the DDSCP Messages

Figure 2 shows the pseudocode for processing an incoming InstallProcess message. The first step is to verify whether the service component trusts the primary authority identified in the message. This is decided based on the policy of the receiving service component. If it does not trust, it sends an InstallProcessReject message to the sender and discards the message without further processing.

```
On Receive InstallProcess Message
  If I do not trust the primary authority
    send  InstallProcessReject and stop processing
  else if Cannot Successfully Verify that
      (Sender is allowed Invocation Authority
       AND  All necessary elements are present
       AND  Integrity of all elements is verifiable
       AND  Message is intended for me
       AND  The Correlation Parameters are allowed
           for the process)
     send  InstallProcessReject and stop processing
  else
    send  InstallProcessAccept
    execute PreProcessActivities if present
    execute the process
END
```

Figure 2. Pseudocode for processing InstallProcess

If the service component trusts the primary authority, it verifies all the signatures on the *InstallProcessHeader* and confirms that the sender of the InstallProcess message is an authorized invocation authority for the present process. It verifies that all the expected elements are present in the message and that the integrity of all the elements is intact. It additionally verifies if it is correct recipient of the process by comparing the identity of the receiving service component specified in the ProcessProperties element.

Lastly, the service component verifies if the authorization of the process to operate on the messages defined by the correlation properties. This may be done based on the rules defined in the policy for the service component. The correlation parameters define a subset of messages, and if admitted the service component only routes the messages defined by that subset to the customized process. In BPEL4WS terminology, a process may not *wait* for any messages outside the subset defined by these parameters.

The main purpose this scrutiny is to ensure that a customized process for one user does not interfere with service for other users. This is done by ensuring that the subsets of "critical" messages for the processes are disjoint. For example if there is process for customized call processing, a process for user TARIQ may only register for messages with TO or FROM field set to TARIQ.

If all of the above verify, the service component sends a IntallProcessAccept message to the sender of the InstallProcess message, and proceeds with the preprocessing steps. If any of the tests fail, the service component sends an InstallProcessReject message to the sender and discards the message without further processing.

The preprocessing activity is a process in itself and the service component processes the activities as usual; however, the preprocessing activities element may contain an XSL transform that takes any preprocessing data elements and the process element as input and generates a new process element. In order to overcome the limitation of XSLT that it takes only one document as input, we must either combine multiple elements into a single or use the entire InstallProcess element as input. Figure 3 shows this process schematically.

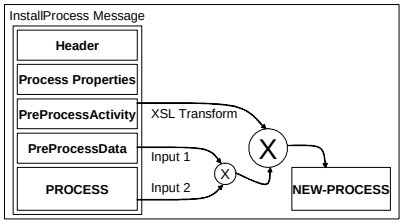

Figure 3: Preprocessing Activities

Upon completion of the transform and any other preprocessing activities, the service component proceeds to execution of the process. The process execution does not require any special attention, besides what is defined in BPEL4WS, until it reaches one of the activities we have defined in previous sub-sections.

The DispatchProcess activity requires the service component to dispatch a process to another service component. The DispatchProcess element includes the process to be dispatched, along with address of receiving service component and a set of PreDispatch Activities. The pre-dispatch activity has same syntax as the preprocess activities; and just like the preprocess activity it may also include a transform that operates on the process element. However, the goal of the pre-dispatch activity is to create necessary pre-process data for the destination service component, and as such, the transform in this activity creates a PreProcessActivitiesData element.

As part of processing this activity, the service component creates an InstallProcess message, copies the Header of the current InstallProcess message to the new InstallProcess. It also copies the ProcessProperties, PreProcessActivities, and Process element from the DispatchProcess activity to the new InstallProcess message. The output of pre-dispatch activity is also spliced into the new InstallProcess Message. The service component now dispatches the InstallProcess message to the destination and continues with any other activities that are included in its process. Figure 4 shows a schematic of the processing of the DispatchProcess activity.

We have already defined the processing of AbortProcess and the Checkpoint activity in previous section.

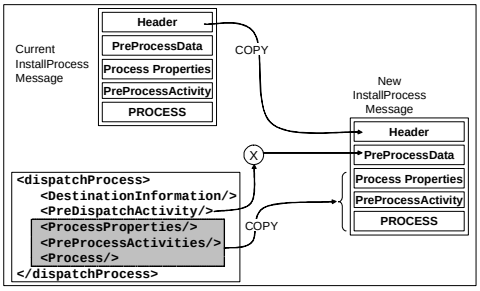

Figure 4: Processing the DispatchProcess Activity

4. Service Platform Model

4.1. Overview

We view every application as a composition of service components. In context of using this platform for services provided by the telecommunication network, the network shall provide some of these service components as part of the infrastructure. Examples of such service components include, network firewall function, authentication, authorization, and accounting functions, billing functions, location information functions, content distribution functions, specialized transport function, e.g., multicast, or privacy enabled transport function, or media translation and processing gateways. Other service components may reside outside the network, at public places, or with third-party service providers, or at user's premises, e.g., his home devices and sensors networks at home.

With dynamic and distributed coordination model, the application initiator dispatches processes to all the service components involved in the application. Upon receipt of these processes, the service components work towards the eventual goal of the application concurrently by executing their assigned processes. The process dispatch protocol groups the related processes together to facilitate the correlation among all the process for the application.

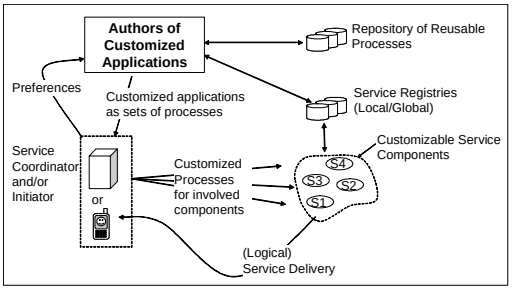

Figure 5. Components of the Service Platform

Figure 5 shows a high-level view of the overall service platform. The authors of the application (programmers or automated agents) provide customized applications to the service initiator or coordinator. The coordinator then disperses these processes to the appropriate service components using the DDSCP. The service components, as a result of execution of these processes, start working towards the application goal. In the course of this execution, the service components may dispatch process to other service components.

The result of the execution of these processes is logically delivered to the user. The platform also includes repositories for registration of available service components and their capabilities, and a repository of reusable processes or patterns, that the application developers can use to expedite the process of service creation further.

One concern with this model is to reduce the number of processes running on the service components. In general, we want to reduce the number of idle or inactive processes. For the services that originate at the user, such as an outgoing call, the user can simultaneously send the signaling message and the active message containing the customized process to

the service components. The process remains active for the duration of the call, and then fades away. However, many services are not initiated by the user requesting the customization, e.g., an incoming call. We have two choices here. First is to "pre-install" the customized processes, however, this is increase the number of inactive processes. Other approach is to have the service components look for appropriate customized process. A repository can maintain all the customized processes, if the process handler does not find a customized process for an a legitimate message, it can query the DDSCP handler to find an appropriate, and DDSCP handler can then trigger the repository to find the best customized process, and dispatch it to the service component. This is shown in Figure 6.

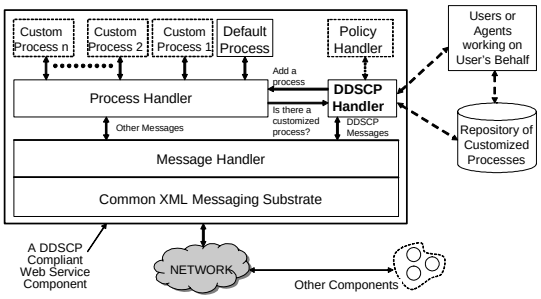

Figure 6. Web Services Component Supporting DDSCP

4.2. New Structure of Web Service Components

Figure 7 shows the stack diagram for dynamic and distributed coordination support for a service component based on web services technologies. It differs from the traditional web services in that it includes a separate layer responsible for dynamic distributed coordination. In particular, this layer works in tandem with the process layer, (shown in figure 6), allowing dynamic installation of new processes on the web service and their proper initialization. These initialization parameters later serve to facilitate correlation between the messages and their intended processes. If the service interface includes methods for DDSCP, it is a declaration that the service supports DDSCP protocol.

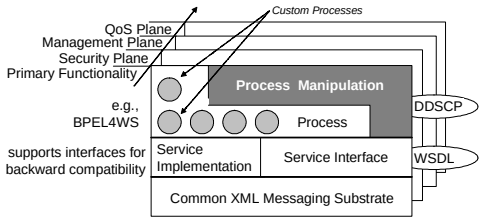

Figure 7: Web Services Component Supporting DDSCP

4.3. Advantages of our model

One advantage of dynamic distributed coordination model is that by installing an appropriate process on a service component, the application can create a customized version of that component. In object oriented programming terminology, this is equivalent to creating an object of a subclass of original service component, with the exception that the subclass in this case cannot introduce new interfaces. The assigned process can override the behavior of the service component within certain constraints. This process of modifying web service components can be referred to as creating customized web-services. In fact, using our model same service component can present different customized behaviors to different applications. Availability of this customizable service component can greatly simplify the application.

Now the question arises, can we not have the process to customize the component reside at the application instead of the component itself?. The short answer is that our model does not preclude this; the customization process can run remotely, if the service component relays all the related messages to the application server. However, having the process locally available at the component has a few advantages. One is that the service component can perform a series of related tasks locally, without reverting to the original application or customization process after completion of each task. This can significantly improve performance of the overall system.

By virtue of DispatchProcess activity, our model allows the service components to dispatch processes to other service components, if the author of the process of the sending service component specifies so in the process. This powerful feature enables dynamic and conditional deployment of new processes, which in turn form new coordination points.

Another advantage is that with our model, the execution of the application does not depend on the persistent availability of the centralized coordinator. Relaxation of this requirement means that now a device like the user's mobile equipment, which may have shaky connectivity, can easily manage its own services.

Yet another advantage of the model is simplification of management of highly customized services. Service customization often means treating the information in unique ways for the customers. If we were to accommodate all the customizable behaviors in a centralized coordination point, the application will become extraordinarily complicated. With our model, the service provider can represent every customized

service for each individual user as a set of customized sub-processes. The complexity of individual sub-processes is much less than that of a centralized process. This division also makes possible the reuse and outsourcing of sub-processes. Moreover, as the service initiator takes the responsibility of "installing" appropriate set of sub-processes on demand, the network and service environment can remain free of inactive processes.

5. Example Usage

5.1. A programmable sensor network controller

Let us start with a simple and non-telecommunication related example. Suppose we have a small sensor network that reads the temperature in different rooms in a house, and controls the windows, doors, and air conditioning system in the house. We can consider the sensor network controller and the controllers of different entities as service components. Let's also assume that only the sensor network controller is "programmable" in the sense that it can receive the dynamic process. Using our model, we can write and dispatch a small process to the controller that tells it what messages to send to doors, windows, and air-conditioning units, based on sensor readings in different parts of the house.

Although this scenario can also be built with other existing models, but with our model, it is much easier to customize the behavior of the sensor network controller.

5.2. Arrangements transcontinental meeting for collaborative work

Consider an application that monitors the calendar service of a user, and makes necessary arrangements for any meetings that are scheduled. The application wants to ensure that there is never a language gap, meeting material is made available to all the participants in appropriate language, that the meeting notes are generated, and all the proceedings are recorded, and that the meeting is arranged through a terminal that most convenient for the user at the time of meeting.

In this example there are a number of service components involved, and the nature of the call that will be established for the meeting is not exactly standard, i.e., it is a customized application that uses the telecommunication infrastructure. Following describes how we can represent this application as a set of distributed processes, running at different service components, so that the network does not need to make any prior arrangements for the service.

The service components involved here are the calendar service (C), a personal assistant service (PA), a user profile information service (Prf), Location service (Loc), the user terminal (Trm), translation service (Trans), recording service (R), and multimedia call signaling proxies (P).

Figure 8 shows the above-mentioned customized call using DDSCP. The circles represent different service components; the circles with shaded cap represent the service components that support DDSCP. A user desiring such an application can obtain it from an external application developer. In this example, we consider that a personal assistance is working on user's behalf, so the user dispatches the process to the PA. The application activities in the process for the process at PA require it to interact with the calendar service to obtain the meeting schedule and participant list.

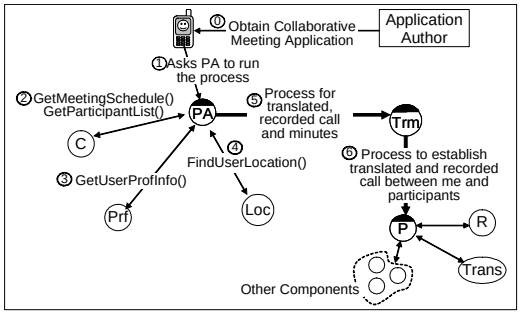

Figure 8: Establishing a Customized Call with DDSCP

At the scheduled time, the PA obtains the profile information about the participants (including information about the languages they speak), and decides whether translation may be required. The PA then finds the location of the user, and the most convenient terminal using the location service. The PA negotiates with the terminal to learn about its capabilities, including whether it supports DDSCP. Based on the terminal capabilities, the PA dispatches a process to the terminal telling it how to establish the desired multimedia session. This process has a number of activities that the terminal has to perform, and a process that it dispatches to the signaling proxy in the network. The process tells the proxy about the session requirements, and the step it needs to follow. It interacts with the translation and recording services to achieve the service goals. Meanwhile the terminal continues to provide interaction with the user. All of this happens concurrently, without any prior provisioning in the network.

5.3. A Note on Implementation

We are developing DDSCP capable web-service components, and a prototype to demonstrate the concept. The prototype closely follows the model of Figure 6, and is being implemented in Java.

However, we have faced some difficulties in using BPEL4WS as the process language. The requirement of defining the roles and partner links, pre-hand in the WSDL is proving to be a major difficulty in using unadulterated BPEL4WS with our service components. We are investigating how to introduce direct addressing, so that we can use the dynamically process easily. Mandell and McIlraith [9] take an interesting approach to dynamically incorporate service partners in BPEL4WS framework. However, we need more than that, because in our model even roles are not predefined. Another difficulty is lack of support to initialize a process externally during implicit starts.

We are exploring how we can make changes that are compatible with existing BPEL4WS framework. We will hopefully report detailed results from the implementation and prototype experience in a future publication.

6. Discussion

Our service platform architecture and DDSCP have a number both interesting and debatable aspects. Let us start by discussing the merits of distributed coordination. It is true that distributed coordination is not beneficial or desirable in all scenarios. For example, in environments where we cannot trust the service components to execute the process correctly and completely, we may want to revert to centralized coordination. Cognizant of this fact, we note that centralized coordination is only a special case of distributed coordination, as defined in our model. Although we presently do not explicitly support dynamically transforming a distributed coordination application into a centrally coordination application, the application author can always use the centralized coordination by avoiding DispatchProcess activities in the application process of the main coordinator, e.g., in the first of the two examples given above. Moreover, a cleverly written preprocessing block can transform the process at runtime such that it uses centralized control for non-trusted service components.

Another point to consider is the need for dynamic deployment and the associated overhead and safety concern. Because of limited number of services offered on networks today, dynamic deployment seems extraneous. However, we believe that need for dynamic deployment will emerge as networks offer more personalized services, especially the kind that are created on demand and have short lifetime. Statically configuring the network infrastructure to *support* all possible customized services is not feasible.

Although our framework caters for process integrity and declaration of correlation parameters to isolate the processes from each other, and minimize chances of interference between processes of different users, the security remains a major concern. There are a number of techniques for portable code, such as Java, that can verify properties, liveliness, access control, and resource for a particular piece of code. However, we realize that security concern in a service platform environment may be different from traditional systems, because of shared infrastructure and highly distributed nature of the system; we plan to address these issues rigorously as part of future work. Admittedly, it is hard to convince a telecommunication network operator, which strives to provide five nines reliability, to opt for something that even remotely resembles a software agent.

Within the context of security and safety, one concern is the presence of transform in the pre-processing activities that has capability to modify the process. While we have taken care to design the system in a way that a service component may execute transforms only on its own process, and that we can ensure the integrity of both the process and the transform during its traversal through other service component, we realize that this is not a elegant solution. Careful authoring of process and allowing initial parameterization can circumvent the need. We are considering a slight modification to our model where we can just “scoop out” the messages identified by the correlation parameters in the InstallProcess message, to a remote service, and run the process at that remote processor.

Within the existing model, in order to address the process transport overhead problem, we are considering how to dispatch processes by reference. We are also investigating whether it possible to safely create customized processes from a basic representation so that we only need to transport the required customization and not the entire process.

Another overhead related with the dynamic deployment of processes is that of increased complexity of individual service components. Considering a non-dynamic deployment scenario; a typical process that capture all the possible customization scenarios, will be large. The way workflows work today, the service component will spawn a new process for every context, e.g., for every

new call created, or an order placed. With dynamic deployment of customized process, although we will have just as many active processes on a service component, as the non-dynamic deployment case, however, customized processes for specific users are likely to be smaller than the all-encompassing process and thus the overall load on the service may be less.

The above argument also related to the point that whether we need “processes” for customizations, or can we just use different parameters in the messages as triggers to customization. We feel that to achieve same level of customization, we will end up with an all-encompassing process that might be both complicated and hard to run and maintain.

7. Conclusion and Future Work

We have presented a technique to enable distributed coordination among components of a dynamically composed web service. This technique has a number of advantages, including performance enhancements, reduced complexity of applications and thus rapid deployment, reduced load on the service environment by allowing users to take charge of their own services, and robustness by eliminating any centralized coordination.

We note that this technique also raises a number of new issues. Most notable of these is that of security and safety. Other issues include improving reusability of processes to enhance rapid service development. Similarly, we have not explicitly addressed the increased complexity of service components themselves, and its impact on scalability. We are addressing these issues as part of on going work.

8. References

[1] D. L. Tennenhouse, D. Wetherall. Towards an Active Network Architecture. In Multimedia Computing and Networking 96, San Jose, CA. Jan. 1996.

[2] D. Wetherall, J. Guttag, D Tennenhouse, ANTS: A toolkit for building and dynamically deploying network protocols. In IEEE Openarch'98. April 1998.

[3] Open Service Access (OSA); Application Programming Interface (API); 3GPP TS 29.198.

[4] The Parlay Group: http://www.parlay.org/

[5] The Parlay Group: Web Services Working Group. "Parlay Web Services Architecture Comparison." October 31, 2002. http://www.parlay.org/

[6] Z. Maamar, Q. Sheng, B. Benatallah . Interleaving Web Services Composition and Execution Using Software Agents and Delegation. AAMAS'03 Workshop on Web Services and Agent-based Engineering, 14 July 2003, Melbourne, Australia.

[7] G. Agha and C. Hewitt. Concurrent programming using actors. In A. Yonezawa and M. Tokoro, editors, Object-Oriented Concurrent Programming, 37-53. MIT Press, 1988.

[8] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klien, F. Leymann, K. Liu, D. Roller, D. Simth, S, Thatte, I. Trickovic, S. Weerawarana. "Business Process Execution Language for Web Services 1.1." May 2003.

[9] D. Mandell and S. McIlraith. "Automating Web Service Discovery, Customization, and Semantic Translation with a Semantic Discovery Service." Poster Session, The Twelfth International World Wide Web Conference 20-24 May 2003, Budapest, HUNGARY

[10] S. Paul, E. Park, D. Hutches, J. Chaar. "RainMaker: Workflow Execution Using Distributed Interoperable Components." ECDL 1998: pp. 801-818

[11] S. Paul, E. Park, J. Chaar. "RainMan: A workflow system for the Internet." USENIX Symposium on Internet Technologies and Systems 1997

[12] S. Song, S. Shannon, M. Hicks, S. Nettles. "Evolution in Action: Using Active Networking to Evolve Network Support for Mobility." IWAN 2002: pp. 146-161

[13] G. Alonso and C. Mohan. Workflow Management Systems: The next Generation of distributed Processing Tools. In S. Jajodia and L. Kerschberg (Eds.): Advanced Transaction Models and Architectures. Kluwer Academic Publishers, 1997, pp. 35-62.

[14] 3GPP TS 22.057. Mobile Execution Environment (MExE);Service description, Stage 1.

[15] 3GPP TS 22.038. USIM/SIM Application Toolkit (USAT/SAT); Service description; Stage 1.