9-1-2010

# A Methodology for Engineering Collaborative and ad-hoc Mobile Applications using SyD Middleware

Praveen Madiraju
*Marquette University*, praveen.madiraju@marquette.edu

Srilaxmi Malladi
*Georgia State University*

Janaka Balasooriya
*Georgia State University*

Arthi Hariharan
*Georgia State University*

Sushil K. Prasad
*Georgia State University*

***See next page for additional authors***

Accepted version. *Journal of Network and Computer Applications*, Vol. 33, No. 5 (September 2010). DOI. 

# A Methodology for Engineering Collaborative and ad-hoc Mobile Applications using SyD Middleware

By Praveen Madiraju, Srilaxmi Malladi, Janaka Balasooriya, Arthi Hariharan, Sushil K. Prasad, and Anu Bourgeois

*Today's web applications are more collaborative and utilize standard and ubiquitous Internet protocols. We have earlier developed System on Mobile Devices (SyD) middleware to rapidly develop and deploy collaborative applications over heterogeneous and possibly mobile devices hosting web objects. In this paper, we present the software engineering methodology for developing SyD-enabled web applications and illustrate it through a case study on two representative applications: (i) a calendar of meeting application, which is a collaborative application and (ii) a travel application which is an ad-hoc collaborative application. SyD-enabled web objects allow us to create a collaborative application rapidly with limited coding effort. In this case study, the modular software architecture allowed us to hide the inherent heterogeneity among devices, data stores, and networks by presenting a uniform and persistent object view of mobile objects interacting through XML/SOAP requests and responses. The performance results we obtained show that the application scales well as we increase the group size and adapts well within the constraints of mobile devices.*

## 1. Introduction

Rapid development of coordinating distributed applications by leveraging off existing web entities is key to bringing the Internet's collaborative potential to the users at large. Such collaborative applications span domains as diverse as personal applications (travel, calendaring and scheduling) to enterprise e-commerce applications (supply chains, work flows, and virtual organizations), and scientific biomedical applications (biomedical data and process integration, and experiment workflows). All the coordinating applications themselves and the constituent autonomous entities are usually hosted on heterogeneous and autonomous, possibly mobile platforms (Krone et al., 1998). There is an emerging need for a comprehensive middleware technology to enable quick development and deployment of these collaborative distributed applications over a collection of mobile (and wired) devices. This has been identified as one of the key research challenges (Edwards et al., 2002; Phan et al., 2002).

*Limitations of current technology*: The current technology for the development of such collaborative applications over a set of wired or wireless devices has several limitations. It

requires explicit and tedious programming on each kind of device, both for data access and for inter-device and inter-application communication. The application code is specific to the type of device, data format, and the network. Managing applications running across mobile devices becomes complex due to lack of persistence data and weak connectivity. A few existing middlewares have addressed the stated requirements in a piecemeal fashion. The current state-of-the-art still lacks in enabling mobile devices with server capabilities, and developing collaborative applications is restricted to few domains. It also suffers in providing group or transaction functionalities and offers limited mobility support. These issues are further elaborated by Prasad et al. (2004).

*Contributions*: Our work is an ongoing effort to address the aforementioned limitations and in Prasad et al. (2004), we reported our first prototype design and implementation of a middleware for mobile devices called System on Mobile Devices (SyD). In this paper, we continue our work and utilize SyD's high-level programming platform to rapidly engineer group web applications over a collection of heterogeneous, autonomous, and possibly mobile data stores. We describe a software engineering based design methodology with case studies on a calendar of meetings application (a collaborative application) (Prasad et al., 2005), and a travel application (an ad-hoc application). We also show that SyD naturally extends to enabling collaborative applications across web-based objects. The SyD objects are stateful, web-based, and provide interfaces like web services for method invocations. Furthermore, all method invocations and their responses in SyD employ SOAP-like XML envelopes. Therefore, SyD objects, their interactions, and the underlying techniques discussed in this paper have a direct bearing on web services and their compositions and coordination, making the development of coordinating applications over mobile devices easier and faster. The middleware addresses the key problems of heterogeneity of device, data format and network, and of mobility. SyD achieves ease of application development, transparency in mobility of code, and the scalability required for large enterprise applications with a small footprint (total of 112 with 76 KB being device-resident) required by handheld devices. SyD also allows creating ad-hoc collaborative applications by composing or configuring pre-existing SyD objects. The application development is both quick and streamlined using a design methodology that includes realizing UML design phases with SyD components (described in Section 4.). We illustrate this with two sample SyD application case studies later in this work; we briefly introduce these two application cases below.

*Sample SyD application case studies*: Currently there are two key SyD-based applications. We implemented these using various technologies, including JDBC, SOAP, and SyD. The SyD-based development was by far the quickest, with more functionalities, due to

high-level APIs of SyD (2–3 weeks each by 3–4 students), with comparable execution efficiencies.

The calendar application is an example SyD application wherein several individuals manage their time schedules. The typical functionalities provided in a calendar application are: (a) set up meeting among individuals with certain conditions to be met such as a required quorum, (b) set up tentative meetings that could not be set up otherwise due to unavailability of certain individuals, and (c) remove oneself from a meeting resulting in automatic triggers being executed. The triggers may possibly convert tentative meetings into confirmed meetings. The calendar application showcases various aspects like constraint-satisfaction in applications to achieve the required quorum, mobility of devices, and heterogeneous data and devices as the individuals maintain their schedules on their devices in a format suitable to them. To implement our calendar application with the current technology involves cumbersome programming, such as opening authorized connections to respective database servers, executing individual queries against several databases and accumulating results of these queries, and manually enforcing constraints (by writing code) that the databases as a whole need to satisfy. Another problem with the current technology is that it is difficult to deal with multiple types of heterogeneity in the representation of time-schedule information. One individual may have different device, data format, or network from another individual. The existing calendar systems also have considerable amount of delays to confirm the availability of all participants and schedule a meeting. In a calendar application, each user has his own database that is either stored locally or on a proxy. The application programming can be logically divided into server side and client side. The server side comprises of all the methods that interact with the local data store and can be invoked remotely. The client side consists of the user interface which enables the user to interact with the application.

The second application of this study is the travel application. In our previous work (Hariharan et al., 2004), we demonstrate a travel application that allows for automatic rescheduling and cancellation of itineraries. Once an itinerary is decided and the trip is planned for the user, corresponding links are created and maintained in the user's database. If part of the itinerary is cancelled, then automatic cancellation of further itinerary schedule occurs. For example in a travel application, if a flight is cancelled, car and hotel reservations are automatically cancelled, thus easing the burden on the user to have to manually cancel all associated reservations. A SyD ad-hoc application developer's nook provides a simple GUI-based interface for the application developer to initially set up and develop SyD-enabled collaborative applications.

The rest of the paper is organized as follows. Section 2 provides a background on the SyD middleware. Section 3 details distributed and ad-hoc collaborative applications. Section 4 describes a design methodology for collaborative applications using a case study on a calendar of meetings application. Section 5 describes ad-hoc collaborative applications design using a case study on a travel application. We present implementation details of an important module relevant to this work in Section 6. Section 7 discusses implementation of a calendar of meetings application and its performance results. In Section 8, we compare our work with the current state-of-the-art. Finally, we make concluding remarks in Section 9.

## 2. SyD Architecture and Coordination Bonds—Background

In this section, we describe the design of System of Mobile Devices (SyD) (Prasad et al., 2003a) and related issues, and highlight important features of its architecture. Each individual device in SyD may be a traditional database such as a relational or object-oriented database, or may be an ad-hoc data store such as a flat file, an EXCEL worksheet or a list repository. These may be residing in traditional computers, in personal digital assistants (PDAs), or even in devices such as a utility meter or a set-top box. These devices are assumed to be independent in that they do not share a global schema and therefore rules out the possibility of unique data representation. The devices in SyD cooperate with each other to perform interesting tasks, and we envision a new generation of collaborative applications built using this SyD framework.

### 2.1. SyD Architecture Overview

The SyD architecture is shown in Fig. 1. SyD uses the simple yet powerful idea of separating device management from management of groups of users and/or data stores. The SyD framework accomplishes distinct management of devices, user data stores and their coordination when needed with its three layered architecture. At the lowest layer, individual data stores such as device data or personal data are represented by device objects that encapsulate methods/operations for access, and manipulation of data (SyD Deviceware). For example, the data objects are high level wrappers for a flat file or an excel file or an XML file. At the middle layer, there is SyD Groupware, a logically coherent collection of services, APIs, and objects to facilitate the execution of application programs. This forms the middleware kernel API. At the highest level are the SyD Applications themselves such as the calendar and travel applications discussed in this paper. The applications rely on middle and lower layer SyD services, and are independent of device, data and network, making the applications appealing to all kinds of heterogeneity.

We have developed a prototype test bed of SyD middleware that captures the essential features of SyD's overall framework. We have designed and implemented a modular SyD kernel in Java. In Fig. 2, we present the internal architecture of the SyD middleware. The SyD Kernel includes the following five modules:

*SyDDirectory*: Provides user/group/service publishing, management, and lookup services to SyD users and device objects. Also supports intelligent proxy maintenance for users/devices.

*SyDListener*: Provides a uniform object view of device services, and receives and responds to clients' synchronous or asynchronous XML-based remote invocations of those services (Prasad et al., 2004). Also allows SyD device objects to publish their services locally to the listener and globally through the directory service.

*SyDEngine*: Allows users/clients to invoke individual or group services remotely via XML-based messaging and aggregate responses. This yields a basic composer of mobile web services.

*SyDBond*: Enables an application to create and enforce interdependencies, constraints and automatic updates among groups of SyD entities and web Services (Prasad et al., 2003b; Prasad and Balasooriya, 2004)

*SyDEventHandler*: This module handles local and global event registration, monitoring, and triggering.

To register services with the middleware, the SyD Application Object Server (shown in the left of Fig. 2) publishes applications with SyDListener. The SyDListener then registers globally with SyDDirectory. In order to call these services, the client user interface (Client UI) makes a remote invocation for some of the application services by invoking the SyDEngine. The SyDEngine then makes a lookup with the SyDDirectoy, and finds the required information for making a remote call such as URL information for the application services. The SyDEngine then eventually makes a remote invocation using TCP/IP on the SyD Application Objects (SyDAppO). SyD applications are likely to be hosted on mobile devices that frequently suffer from intermittent disconnections or battery discharges. The SyD framework therefore provides tolerance for disconnected devices through its proxy. Extended details regarding the SyD middleware can found in Prasad et al. (2003a).

A key goal of SyD is to enable SyD objects to coordinate in a distributed fashion, possibly in an ad-hoc way. Each SyD object is capable of embedding SyD coordination bonds to other entities enabling it to enforce dependencies and act as a conduit for data and control flows. Over data store objects, this provides active database like capabilities. In general, aspect-oriented properties among various objects are created and enforced dynamically. Its use in rapid

configuration of ad-hoc collaborative applications, such as a set of calendars for a meeting setup (Prasad et al., 2003b), or a set of inter-dependent web services in a travel reservation application (Hariharan et al., 2004), has been demonstrated. The SyD bonds have the modeling capabilities of extended Petri nets and can be employed as general-purpose artifacts for expressing the benchmark workflow patterns (Prasad and Balasooriya, 2004, 2005)

### 2.2. SyD Coordination Bonds

Coordination bonds enable applications to create contracts between entities and enforce interdependencies and constraints, and carry out atomic transactions spanning over a group of entities/processes. The constraints and dependencies can be of QoS type like budget, deadline, etc., or member dynamics such as, all inclusion (and), exclusion (xor), any inclusion (or), etc., or any user defined constraints. While it is convenient to think of an entity as a row, a column, a table, or a set of tables in a data-store, the concept transcends these to any SyD object or its component. There are two types of bonds: subscription bonds and negotiation bonds. Subscription bonds allow automatic flow of information from a source entity to other reference entities that subscribe to it. This can be employed for synchronization as well as more complex changes, needing data or event flows. Negotiation bonds enforce dependencies and constraints across entities and trigger changes based on constraint-satisfaction. SyD bonds may be further combined with other constraint logics like and, or, xor, which are user defined (Joshi, 2005).

A SyD bond is specified by its type (subscription/negotiation), status (confirmed/tentative), references to one or more entities, triggers associated with each reference (event-condition-action rules), priority, constraints (and, or, xor), bond creation and expiry times, and a waiting list of tentative bonds (a priority queue). A tentative bond may become confirmed if the awaited confirmed bond is destroyed. Let an entity *A* be bonded to entities *B* and *C*, which may in turn be bonded to other entities. Under subscription bond logic, a subscripted change in *A* may trigger changes in *B* and *C*, and under negotiation bond logic, *A* can change only if *B* and *C* can be successfully changed. In the following, the phrase ‘‘Change X’’ is employed to refer to an action on *X* (action usually is a particular method invocation on SyD object *X* with specified set of parameters); ‘‘Mark *X*’’ refers to an attempted change that triggers any associated bond without an actual change on *X* (Balasooriya and Prasad, 2005; Prasad and Balasooriya, 2005).

*Subscription Bond*: Mark A; If successful, Change *A* then Try: Change *B*, Change *C*. A ‘‘try’’ may not succeed.

*Negotiation-and Bond*: *Change A* only if *B* and *C* can be successfully changed. (Implements atomic transaction with ’’and’’ logic.)

*Semantics* (may not be implemented this way):

*Mark A* for change and *Lock A*
If successful
*Mark B* and *C* for change and *Lock B* and *C*
If successful
*Change A*
*Change B* and *C*
*Unlock B* and *C*
*Unlock A*

Note that locks are only for the explanation of the bond semantics. A reservation/locking mechanism to implement this usually will have an expiry time to obviate deadlocks. In a database web service, this would usually indicate a ‘‘ready to commit’’ stage.

*Negotiation-or Bond*: Change *A* only if at least one of *B* and *C* can be successfully changed. (Implements atomic transaction with ‘‘or’’ logic and can be extended to *at least k out of n*.)

*Semantics*:

*Mark A* for change and *Lock A*
*Mark B* and *C* for change; Obtain locks on those entities that
can be successfully changed.
If obtained at least one lock
Then *Change A*; Change the locked entities.
Unlock entities

*Negotiation-xor Bond*: Change *A* only if exactly one of *B* and *C* can be successfully changed (implements atomic transaction with ’’xor’’ logic and can be extended to *exactly k out of n*).

*Semantics*:

*Mark A* for change and *Lock A*
*Mark B* and *C* for change. Obtain locks on those entities that can be successfully changed.

If obtained exactly one lock
Then *Change A*; Change the locked entities.
Unlock entities

*Notations*: A subscription bond from *A* to *B* is denoted as a *dashed directed arrow* from *A* to *B*. A negotiation bond from *A* to *B* is denoted as a *solid directed arrow* from *A* to *B*. A negotiation—and bond from *A* to *B* and *C* is denoted by two solid arrows, one each to *B* and *C*, with a ‘‘*’’ in between the arrows. Similarly, a negotiation—or bond from A to B and C is denoted by two solid arrows, one each to B and C, with a ‘‘+’’ in between the arrows. A negotiation-XOR bond from A to B and C is denoted by two solid arrows, one each to B and C, with a ‘‘^’’ in between the arrows. A tentative bond, which is a negotiation bond in a waiting list, is shown as a solid arrow with cuts.

A negotiation bond has two interpretations: pre-execution and post-execution. In case of pre-execution, in order to start activity *B*, *A* needs to complete its execution. In case of post-execution, in order to start activity *B*, *B* needs to make sure that A can be completed afterwards. In this paper, we have primarily employed the pre-execution type of negotiation bonds.

## 3. Collaborative Applications—Distributed and ad-hoc

In this section, we introduce two kinds of collaborative applications, and illustrate them with our case studies on a personal system of calendar application and travel application used as case studies throughout the paper. Both distributed applications of personal system of calendar application and ad-hoc travel application can be easily developed using the SyD framework and showcase SyD capabilities effectively.

### 3.1. Collaborative SyD Applications

Collaborative group applications leverage off multiple constituent web entities, where each of those entities is a server application/component or an object or a data store. A centralized coordinator application resides on one host and composes or configures multiple SyD objects (which may themselves be typically distributed). Composition is achieved through method calls of constituent objects. Configuration additionally employs the SyD coordination bonds to establish flow and dependency structure between coordinator application and constituent objects. These get triggered at various points in execution of coordinator application.

*Centralized vs. distributed coordination*: Not much work has been done in the area of web

service composition for small mobile wireless devices. Disconnection and memory constraints are two important issues considered while designing any application targeted for small handheld devices. Chakraborty et al. (2004) survey the issues related to service composition in mobile environments and evaluate criteria for judging protocols that enable such composition. It states that many of the current technologies, still, do not cover all these aspects in their implementation. Some of the proposed approaches that handle centralized coordination of web services suffer from central point of failure despite making the design and implementation simple. As opposed to the prevailing centralized coordination, distributed coordination has the following two advantages: (i) due to security, privacy, or licensing imperatives, some web-based objects will only allow direct pair-wise interactions without any coordinating third-party entity; and (ii) centralized coordination/ workflows suffer from issues such as scalability, performance, fault tolerance, etc. Achieving coordination in collaborative applications consisting of composed web services for mobile environment is still an evolving area and much work needs to be done. A distributed coordinator application primarily employs SyD bonds among constituent SyD objects and thus is co-hosted distributively.

*Ad-hoc Applications*: Ad-hoc SyD applications leverage off preexisting objects and typically create coordinated application by simple composition of constituent objects or simple configuration using SyD bonds. An ad-hoc application allows web-enabled objects to find services of common interest and compose them to suit the application need. The composition and integration of these objects may vary from being simple (without any constraints enforced) to complex (with pre-defined constraints among the objects). The constraints can be defined over a group of users, objects and/or applications. We refer such a collection of group dependency objects as an ad-hoc group SyD object. All SyD objects are autonomous objects that can communicate in a distributed, peer-to-peer fashion and can be made web-enabled. SyD provides a way to build on-the-fly applications by a proper composition and integration of the pre-existing SyD objects for simple applications and additionally configuring SyD bonds for complex applications. SyD gives a methodology to configure SyD objects on-the-fly via an ad-hoc application development. We give a design methodology and calendar specific details of the development methodology in Section 4.

Garbinato and Rupp (2003) define that ad-hoc applications meet three essential criteria of: (i) mobility, (ii) peer-to-peer, and (iii) collocation. SyD applications reside on mobile, heterogeneous, and autonomous devices giving application level mobility. SyD enabled mobile devices can serve both as a client/server to any service. SyD users communicate with each other in a peer-to-peer fashion. By the definition of collocation, the application is

proximity-restricted and has to end up in a physical transaction. SyD supports both logical and physical proximity based applications. The SyD applications qualify with these criteria.

### 3.2. A Calendar of Meetings Application

A calendar of meetings application illustrates a distributed coordinator application. Prasad et al. (2003b) demonstrated how an empty time slot is found, how a meeting is setup (tentative and confirmed), and how voluntary and involuntary changes are automatically handled. We now provide an overview here. A simple scenario is as follows: A wants to call a meeting between dates d1 and d2 involving B, C, D and himself. After the empty slots in everybody's calendar are found, a ''negotiation-and bond'' is created from A's slot to the specific slot in each calendar table shown as solid lines (Fig. 3).

Choosing the desired slot involves an attempt to write and reserve that slot in *A*'s calendar, triggering the negotiation—and bond. The sequence of *actions* of this bond is to: query each table for this desired slot, ensure that it is not reserved, and reserve this slot. If this sequence of actions succeeds, then each corresponding slot at *A*, *B*, *C* and *D* create a negotiation bond back to *A*'s slot. Else, for those individuals who could not be reserved, a tentative back bond to A is queued up at the corresponding slots to be triggered whenever the status of the slot changes. Assume that *C* could not be reserved. Thus, *C* has a tentative bond back to A (shown as solid line with dashes), and others have subscription bond, shown as dotted line, to A (Fig. 4). Whenever *C* becomes available, if the tentative bond back to *A* is of highest priority, it will get triggered, informing *A* of *C*'s availability, and will attempt to change *A*'s slot to be confirmed. This triggers the negotiation—and bond from *A* to *A*, *B*, *C* and *D*, resulting in another round of negotiation. If all succeed, then corresponding slots are confirmed, and the target slots at *A*, *B*, *C*, and *D* create negotiation bonds back to *A*'s slot (Fig. 4). Thus, a tentative meeting has been converted to confirmed. Now suppose *D* wants to change the schedule for this meeting. The reschedule meeting process happens automatically in real time. A reschedule request from *D* triggers its' back bond to *A*, triggering the forward negotiation—and bond from *A* to *A*, *B*, *C*, and *D*. If all succeed, then a new duration is reserved at each calendar with all forward and back bond established. If not all can agree, then *D* would be unable to change the schedule of the meeting.

### 3.3. An ad-hoc Travel Application

The technologies with Web services have progressed a long way now and have become more sophisticated, interconnected, and interoperable. A travel application can integrate the

reservations of flights, rental cars, and hotel accommodations. Most existing travel reservation applications do not combine and maintain a global relation among these services. As a result, manual changes need to be performed if one portion of the itinerary changes. The process behind such applications would not only integrate these Web services, but also enforces QoS constraints, such as deadlines, budgets, etc. This application integrates the reservations of flights, rental cars, and hotel accommodations using Web Bonds (further explained in Section 5). The development of this application exploits the rapid application development feature of SyD. SyD coordination bonds in web-enabled SyD objects serve as web bonds. By leveraging off existing web services, the developer needs only to select the desired services via UDDI and include the required global logic to link the chosen services.

As mentioned above, the travel application allows for automatic rescheduling and cancellation of itineraries. Once an itinerary is decided and the trip is planned for the user, bonds that are created are maintained in the user's database. This way, the itinerary is still ''alive,'' meaning there is a global relation over these web services and thus providing 'statefulness' to the web services. Any changes made in any one of the web services will affect the other web services associated with that current service. If the flight is cancelled, then automatic cancellation of car and hotel reservations will be triggered, thus easing the burden of the user to manually cancel all associated reservations.

## 4. Designing Collaborative Applications

In this section, we give a methodology for designing collaborative application using concepts of SyD.

### 4.1. Methodology

SyD middleware provides components to aid easy development of collaborative applications that span from centralized to pure distributed. Collaborative applications interact with each other and in the process may encounter data dependencies, control dependencies, or both depending on the nature of the application.

The SyD components provide an effective way of collaboration with heterogeneous peer devices and also provide a way to enforce dependencies. SyD bonds provide methodologies to enforce data and control dependencies in such application scenarios. The challenge is to associate SyD bonds in an early stage of application design for its effective use. In fact, one can follow standard UML design methods to design applications (Fowler and Scott, 2002) and then insert SyD artifacts at appropriate design phases as required. We will explain the design process

of a collaborative application using SyD middleware and SyD bonds based on UML (Pressman, 1997).

The sequence of steps for designing distributed applications using the concepts of SyD is (captured in Fig. 5):

*Step 1*: As an initial step, requirement specification is given by the user of the application system describing the way the system is expected to work.

*Step 2*: A requirement analysis is carried out to identify actors and use cases. An actor is an external entity (person, another system, or object) that uses the system. Use cases are either text descriptions or flow descriptions of how actors interact with the system in all scenarios encountered in the applications. From use cases and actors, use case diagrams are drawn. Use case model diagrams show interaction between actors and all use cases.

*Step 3*: Based on the derived use case diagrams, use cases and actors from Step 2, activity diagrams are developed. UML activity diagrams are equivalent to flow charts and data flow diagrams in object-oriented paradigm. In activity diagrams, the data flow spans across use cases and allows one to identify data and method inter-dependency of the use cases at an abstract level. These data and control dependencies can be analyzed, attributed as SyD-bondable and may be realized using SyD bonds in the later step.

*Step 4*: The identification of classes and class diagrams follows activity diagrams. Class diagrams represent the static behavior of the system. Class diagrams describe the object types in a system and their relationships. Class diagrams model class structure and its contents using design elements such as classes, packages, and object. The persistent or non-persistent data objects with dependencies can be modeled using SyD methods to automate any method invocation needed for the application. Dynamic behavior of the system is modeled using sequence diagrams and collaboration diagrams. Both these diagrams help to identify inter-service dependencies at method level where we can apply SyD bonds to enforce them. Such design can further be clarified using communication diagrams that show the message flow between objects.

Once all the objects, data, data dependencies, and control dependencies have been identified and modeled using SyD and other components, implementation can begin. Server logic can be coded starting from SyD-listener skeleton which is middleware specific. Client coding can be started using SyDBond, SyDEngine, SyDDoc directory logic which is application specific. Fig. 5 shows our collaborative application design process.

### 4.2. Designing Calendar Application—A Case Study

Here, we illustrate the design process with a distributed calendar application. We will limit the discussion to particular scenarios in the system wherever appropriate.

*Step 1*: The requirements specification details the view of user and addresses the aspects of the benefits of the new system, interaction with other systems and system functionality. This includes the details of available time slots, its representation, need to collaborate before deciding on a place and time to meet, any constraints to be met, etc. Based on the specification, several different use cases are identified for calendar application. The use cases of interest are: *get available times, setup meeting, cancel meeting, view calendar, reschedule meeting, create bond,* and *delete bond*.

*Step 2*: The actors and its interactions are then modeled as use case diagrams. The text description of the cancel meeting use case is given in Table 1. The interaction between the actors and all use cases of the system can be given in a use case model diagram.

*Step 3*: We extend use case diagram of cancel meeting to the activity diagram for cancel meeting. For the calendar application, the method call for *cancel meeting checks* for any dependencies associated in its execution (see Fig. 6).

As shown in Fig. 6, dependencies are managed using bonds and deleting corresponding bonds make sure that all required attendees agree on the cancellation. The presence of confirmed dependencies will result in its successful execution. However, in case of tentative dependencies, a reschedule is triggered resulting in an automatic execution of the scenario ‘‘conversion of status’’, in case of no conflicts. These method dependencies indicate placeholders for SyD methods (Prasad and Balasooriya, 2004, 2005).

*Step 4*: The methods *cancel meeting* (attendeelist, starttime, and endtime), *reschedule* (attendeelist, starttime, and endtime), *confirm meeting* (attendeelist, starttime, and endtime), etc., when executed in the calendar application result in the update of dependent data objects. These data dependencies indicate placeholders for SyD bonds. The identified objects and dependencies that can be enforced using SyD bonds are identified in the resulting class diagram.

## 5. Designing ad-hoc Collaborative Applications

SyD allows rapid development of a range of portable and reliable collaborative applications, including ad-hoc applications by the users. It provides well-defined steps and a layered middleware environment to quickly develop applications by composing and bonding existing and new constituent objects. Within this section, we will describe the design process a

user will employ to develop such an ad-hoc application. The development procedure involves using the developer's nook, which is the SyD ad-hoc application development environment. We will also explain the detailed steps performed in developing the travel application (our second case study application).

### 5.1. Ad-hoc Application Design

Users can develop and deploy ad-hoc collaborative applications on-the-fly by leveraging off the pre-existing providers of services/methods and data sources, SyD Client Objects (SyDCOs), SyD Middleware Objects (SyDMWOs) and SyD Application Objects (SyDAppOs), by composing the SyDCOs and non-SyD objects in an application-specific structure through SyDBonds as follows:

(a) Search and locate the required SyDCOs and other objects by employing the SyD Directory Service.

(b) Develop and deploy the ad-hoc SyD collaborative application (SyDAppO) by employing a suitable domain-specific GUI-based SyD ad-hoc application developer service as follows:

1 Choose the desired SyDCOs and other objects to be part of this ad-hoc application SyDAppO.

2 Create SyD bonds among the SyDCOs and other objects and define the attributes of each bond thereby establishing the required constraints and dependencies among the constituent objects (or their parts), and verify the intended functionality and QoS attributes by a simulated execution.

3 Launch the ad-hoc application SyDAppO thereby registering it with the SyD Directory Service.

Register the resulting sydgroups and applications so that further applications may be built.

All the devices with SyD middleware installed on them, can host SyD enabled applications called SyDAppos. To run an application, the application is launched on a home SyDMW (each application may have a home SyDMW providing specialized SyDMWOs), and registered with the home and other SyDMWs. A typical user may run this application by joining the group of users executing on the home SyDMW. Another option is that client's SyDMW downloads the application and launches it. The third scenario is that the client's SyDMW runs the application locally, but employs the home SyDMW for its special services. As can be seen, the

process of developing SyD applications is highly distributed. Data stores are accessed via SyDCOs that encapsulate them, SyDAPPOs coordinate the collection of SyDCOs involved in the application, and SyDMW provides the various services that enable group and communication primitives. This highly distributed approach to application development creates flexibility and ease of programming, allowing rapid development of applications.

### 5.2. Designing Travel Application: A Case Study

As mentioned in Section 3.3, the travel application integrates the reservations of flights, rental cars, and hotel accommodations by using SyDBonds. The process behind this application is the web services framework, so that the application not only integrates these existing web services, but also enforces QoS constraints, such as deadlines and budgets.

The centralized coordinator of the travel application is the SyDBond module. It is the module that invokes the appropriate methods and web services required. SyDBond also automatically triggers requests to web services that are interlinked. It is therefore in-charge of wrapping the method invocation into SOAP requests, and getting back the response and returning only the desired result back to the user. The travel application takes advantage of the rapid application development feature of SyD. By leveraging off of existing web services, the developer needs only to select the desired services (by providing the WSDL) and include the required global logic to bond the chosen services.

Rather than creating distributed bonds that are maintained across each of the web service objects, the SyDBond module can serve as a central coordinator. This is implemented in our travel application. The web services that are collaborating with our travel application are legacy web services and are not SyD enabled. For this reason, these services are not capable of coordination bonding between the various entities. To enable legacy services with coordination bond logic, the SyDBond module wraps the method invocations into SOAP requests, which can then be processed by the legacy web services.

Since travel application involves negotiation bonds among various phases of its itinerary, we need a mechanism to enforce dependencies and constraints across entities and trigger changes based on constraint-satisfaction. The non-SyD-enabled web services will not be able to perform the constraint checking and automatic triggering. By centralizing this control across the entities at the SyDBond module, we are able to interlink existing web services together with the coordination of multiple bonds. The bond module holds all associated methods for a particular service. As shown in Fig. 7, if one service is cancelled, this will automatically trigger the deletion methods on all associated or interlinked services.

Just as the SyDBond module is responsible for coordinating particular methods across entities and automatically triggering methods on services that are bonded together, it is also responsible for enforcing specified constraints. Fig. 8 shows how the SyDBond module is the centralized coordinator for checking these constraints. The bond module holds all associated methods for a particular service. The enforcement of constraints is performed across all entities that are interlinked, but it is performed through the communications invoked by the SyDBond module.

## 6. Implementation of SyDBond Module

In the previous section, we have presented centralized coordinating and constraint checking across multiple applications using SyDBond module. The implementation details of SyDBond module and SyD database are explained in this section. We provide the details to set up the SyDBond database, its tables, and how to identify and associate SyDBond methods. We also show the procedure for initial set up of the travel application by using the developer's interface. The developer's interface is simple and menu driven. With just a few clicks, one can setup the necessary SyDBond database, initialize tables, view the services available, and specify constraint logic. The developer simply needs to specify the database username, password and the jdbc connection string and the tables are then setup automatically for the user.

### 6.1. Initial Setup of SyDBond Database

All the information concerning a bond is maintained in a bond database that is stored locally by the user. This bond database is created for a user when he/she installs a SyD application with bond-enabled features. As of now, SyDBond is compliant only with Oracle database.

Some of the important tables that are created include:

- *syd_bond*: main table that holds the details of bonds
- *bond_method*: holds the method names and corresponding names of methods to be triggered upon automatic updates, cancellations, etc.
- *waiting_bond*: holds the waiting bond details
- *service_info*: stores details of web services, its wsdl URL and methods
- *global_constraint*: used for maintaining global constraints such as budget, deadlines, etc.

*Population of SyDBond tables*: This is an important event of the initial set up. The application developer gives details of the web services that are to be included in the application.

The details include the service name, the wsdl url, and optional features such as the list of methods that can be invoked and an associated priority. Once the developer feeds in the data, the *parsewsdl* method in SyDBond may be invoked. This method parses the specified wsdl file and stores the lists of methods that are listed in the web service. All the required information is then stored in the *service_info* table of SyDBond database. The developer has an option to view *service info* table any time and see the list of web services available, add or delete entries within the table in an easy manner.

Enforcing constraints and interdependencies are some of the vital features of SyDBond. In order to achieve this, the *constraint* and *bond_method* tables have to be populated. The *constraint* table holds information as to which methods of the web services are related with constraints such as budgets, deadlines. Likewise, the *bond_method* table holds the list of methods and their associated methods to be triggered. This information is used for automatic triggering on events in case of cancellations or reschedules. Once the SyDBond tables are populated we can use the methods of SyDBond to develop new applications in an ad-hoc manner.

### 6.1.1. Using SyD Developer's Nook for Initial Set up

The SyD middleware provides a simple GUI-based interface for the application developer to initially setup and SyDBond-enable his/her application. We refer to this interface as the *Developer's Nook*, as it provides a separate working area for the application developer. As SyDBond attempts to make things as automated as possible, the developer needs to initialize certain entities based on the business logic. He/she is given access to different GUI screens to perform various functions such as setting up database tables, populating table values, specifying constraints and methods that qualify for auto-triggering. We will go through in depth details for selective functions.

The application developer can also specify the constraints on web services that are to be interlinked using the SyDBond module. The developer specifies a service name (e.g. Flight Service) and gives its wsdl URL (e.g. http://www.xmethods.net/sd/2001/Flight Service.wsdl). The developer also has an option to enumerate the methods in the service. This in turn, invokes methods of SyDBond that parses the wsdl. If the location of the URL is faulty, an appropriate error message is thrown.

The application developer can specify constraints associated with methods. This is also done using a simple GUI. When the developer chooses the desired method and the associated constraint, entries are made in the user's database. This information is later used by the

SyDBond module to check for constraints associated with methods. Upon service method invocation through SyDBond, it checks to see the associated constraints on the methods. The application developer also chooses the methods that are to be bonded. These details are stored in the database of the user. SyDBond, upon any method invocation checks to see for associated methods that are to be triggered. It then automatically invokes the rest of the methods that are bonded.

After the initial set up is complete, the application developer can finalize the details pertaining to the application. In the travel application, this includes creating the login page and a main page that offers the appropriate options that include viewing one's itinerary, making/rescheduling one's reservation or viewing their set up page.

For instance, the user makes reservations by selecting the necessary itinerary entities. The user also has an option to specify any constraints that needs to be considered such as budget, time, etc. Once the user gives all these details, SyDBond first packs the details in a SOAP request, branches out to various web services and invokes the corresponding methods. It then returns the appropriate results back to the user. Once the user has decided on the itinerary, after the confirmation, SyDBond then forms the bonds for this itinerary. The bonds are then ''live'' meaning that any change in any one of the entities of the bond, causes an automatic affect on the other entities.

When itineraries are displayed, the user has an option to cancel his/her itineraries. When the user chooses any one of the segments to be cancelled, automatic cancellations of the rest of the trip is done by SyDBonds. When user chooses to cancel, say, user's flight reservations, SyDBond checks to see if there are any auto triggers associated with that method. It checks in the database for associated methods to be triggered. Once the methods are identified, it automatically invokes the rest of the methods and all bonded segments of the itinerary are cancelled.

### 6.2. Significant SyDBond Methods

As we have seen, the effort taken by the application developer to develop an application is minimal. An application developer needs to mainly participate only in the initial set up and developing the GUI. Listed below are some of the important SyDBond methods that help to accomplish this ease of development of an application:

- *createSyDBondDatabase*: This method is invoked to create all the necessary tables of SyDBond. This call is done initially when the application developer needs to make an application SyDBond-enabled.

- *createSyDBond*: This method is used to form associations or bonds among entities. When a schedule is decided upon, bonds are created. Details like source entity, destination entity, start time, expiry time, constraints, priority, comments, etc., are specified and a bond between the source and the destination entities is created. These details are later used by the SyDBond module for automatic reschedules, and updates.
- *parseWsdl*: This method parses the wsdl file of the web service, lists out the methods that can be invoked, parameters to be used, etc. The application developer initially, gives the url location of the web services which he desires to integrate. This method is then invoked to parse the wsdl. A DOM parser is used in this method to parse the XML document. Methods names (and their parameter types) of the given web service are then extracted and placed in a table for further reference.
- *packAndSend*: This method is used to invoke methods of web services. This creates the SOAP envelope by packing the necessary parameters and sends the request to the web services. When a SOAP response it obtained, it then returns the desired output back to the user.
- *packAndSendConstraints*: This method is also used to invoke methods of web services. This creates the SOAP envelope by packing the necessary parameters and sends the request to the web services. However, methods that are associated with constraints like budget are executed through this method of SyDBond. The resulting response is aggregated and the only results that satisfy the constraints are returned.
- *viewBonds*: This method is used to view all the bonds associated with a particular user. This is a simple yet useful method of SyDBond. In case of a travel reservation application, upon this method invocation, would result in the itineraries being displayed.
- *autoTrigger*: Before any method is executed, it is first checked to see if there are any methods coupled with it that need to be triggered. This method of SyDBond is used to realize it. Multithreading of methods is employed to achieve faster execution time.
- *checkOnWaitingBonds*: This method is invoked upon any bond deletion. A check is done to see if there are any waiting bonds associated with the bond currently being deleted. If there is such a case, then the waiting bond is converted to a permanent bond and an entry is made in the bond table.
- *deleteBond*: This method is invoked upon any bond deletion. A check is done to see if there are any associated bonds to be deleted (using autoTrigger method) and the bonds are physically removed from the database.

### 6.3. An Example Usage of SyDBond Module

Here we discuss an example usage of SyDBond module focusing on the cancel meeting scenario in the SyD Calendar Application. This description will help to highlight some of the methods of the SyDBond module and show the interaction of it with the bond database and other modules of SyD Middleware. The Calendar application is dependent on SyDBonds in order to manage the interdependencies between various calendars. Cancel meeting especially involves following all the interdependencies and automatically converting a tentative meeting to permanent based on priority. Using SyDBonds the application can call deleteBond( ), which follows the following steps to achieve automatic triggering.

1. Check to see if there are any associated waiting bonds.
2. If so, automatically convert status of waiting bonds from tentative to permanent through SyDEngine.
3. Delete the local bond.
4. Invoke deleteBond on the rest of the associated bonds.
5. Update the calendar database of the user.
6. SyDEngine gets the remote URL of the associated users from the SyDDirectory Service and invokes the necessary method.
7. Repeat steps 1 through 6 for each associated user.

## 7. Calendar Application Implementation and Experiments

In this section, we discuss implementation and experiments on the calendar of meetings application. The performance metrics like response time, server processing time, etc., for various meeting scenarios are evaluated and compared.

### 7.1. Calendar Application Implementation

The design of the calendar application has been implemented on HP iPAQ H3600 and H3700 series running windows CE operating system. Here, we describe implementation details providing insights into the development process. These development logistics and device-level details should help developers of similar applications for mobile devices.

*Step 1*: We implemented SyD Middleware (as a Java package) and Calendar code using Java JDK 1.3. The system user interface was designed using Java Applets. We used Oracle8i as the back end database for storing SyD bond and application specific tables. All were implemented on a PC. Calendar application code interfaces with SyD Middleware application

code for executing method calls (SyDEngine), listening for incoming method calls (SyDListener), and making directory service calls (SyDDirectory).

*Step 2*: We installed JVM for iPAQ, Jeode EVM Version 1.9. We ported the SyD Middleware code and calendar application code on the iPAQ using Microsoft ActiveSync version 3.5 and set the classpath appropriately.

*Step 3*: After downloading the SyD Middleware, we installed and ran the middleware components on the iPAQ. This involves: (i) running a directory server (Oracle server) on a PC connected via a wireless network with the base iPAQ and (ii) running *listener.lnk* file (located in/syd/sydlistener path), which continuously listens for incoming method calls.

*Step 4*: We then installed the calendar application code itself. To do this, we executed the *CalRegistrar.lnk* file, which registers the application with SyDDirectory, followed by the application GUI to implement the various scenarios (set up meeting, cancel meeting, and reschedule meeting).

### 7.2. Experiments and Performance Metrics

We ran our experiments on a high performance/low power SA1110 (206 MHz) Compaq iPAQ H 3600 and 3700 series, with 32 MB of SD RAM and 32 MB of Flash ROM. We had three 3600 series and seven 3700 series iPAQ running middleware and calendar applications connected through a wireless network using a 2.4 GHz wireless router. The operating system was Windows CE. We used JDK version 1.3 to code our programs and JVM for iPAQ was Jeode EVM Version 1.9. The DBMS of the directory server was Oracle 8i.

In Section 4 we have shown that SyD middleware enables structured, streamlined and rapid application development on mobile devices backed with theoretical and proven case study implementations of the calendar application. However, in a mobile setting, it is also significant that the applications developed scale well in terms of bandwidth, memory storage and response time parameters, as these resources are scarce for mobile devices. The motivations for considering aforementioned parameters are as follows: (1) mobile devices cannot afford large amounts of message transfers, as the network bandwidth is limited; hence, we measured message size transferred; (2) storage size on iPAQ is scarce and larger storage size for applications is not desired; hence, we measured storage requirements; (3) response time for executing method calls on mobile devices is critical, as higher response times are possible when applications: (a) consume more storage space, (b) transfer larger message sizes, and (c) require higher memory; hence we measured response time. We carried out experiments on calendar application for three scenarios: set up meeting, cancel meeting, and reschedule meeting. Our

experiment results have been encouraging, as the application has shown to scale well in terms of all the parameters.

### 7.2.1. Setup Meeting Scenario

A constant message size of 50 bytes is transferred for each participant in a meeting consisting of meeting details. The storage size for group sizes of 2, 3, and 4 are: 120, 146, and 170 bytes, respectively. For group sizes of more than 3, the storage size does not increase linearly as we associated a meeting id for each meeting, which avoids repetitive information such as start times, end times, and comments.

*Response time*: Response time is the time required to execute set up meeting method call. A set up meeting method call includes time required to execute a get available time method returning the available times of all the participants, time required to execute the set up meeting for all involved meeting participants, and time to write the meeting details of all the participants to a file. It should also be noted that any method call must go through SyD middleware components. More specifically, it includes time required for (i) SyDEngine to contact SyDDirectory to get other user url information, (ii) SyDDoc to create a request document, and (iii) SyDEngine to invoke SyDListener remotely and get back the results.

In Fig. 9, we show the response time for all three scenarios based on varying number of group sizes. We observe that response time scales well (does not increase rapidly) for increasing group size through parallelism in processing and this behavior can be explained by analyzing different middleware component timings that make up response time as can be seen from Fig. 10. The different components and their timing analysis are given below:

The ‘‘Engine to Directory Service’’ takes around 47–60 ms for group sizes of 2–10, which is less than 1% of total time. The ‘‘Create SyDDoc’’ value ranges from 13 to 90 ms for group sizes of 2–10, which is again less than 1% of response time. Now, we go in details on the components that make up a large share of the total response time.

*Engine to remote listener*: SyDEngine invokes remote listener for executing method call on remote devices by using the request document generated from the above step. This involves sending the request document to the remote listener, parsing the request document at the remote listener end, invoking the method call on the remote listener and writing the meeting details of each individual participant to a file. For increased group sizes, we achieve some concurrency as multiple remote listener calls are made to participant devices and results are collected. This value ranges from 1725 to 2900 ms for group sizes of 2–10 (takes around 48% of total time).

*Server processing*: This refers to all other miscellaneous processing times such as opening, writing, and closing of file at initiator side, initializations for middleware components (SyDEngine, client side RMI registry components of directory server), and different application specific objects such as vectors. Here, we achieve concurrency for increased group sizes. This value ranges from 1995 to 2100 ms for group sizes of 2–10 (takes around 50% of total time).

#### 7.2.2. Other Meeting Scenarios

In a reschedule meeting scenario, from the initiator point of view, size of message transferred is the message size transferred to convey the information that meeting has been cancelled to the other participants, and another message to send a confirmation of the meeting set up that has been tentative so far. The initiator does not have to wait on any acknowledgements in either case as one corresponds to cancel and for the tentative meeting the timings have been already agreed as tentative. We assume that only an initiator can cancel the meeting as he alone knows all the participant details and the tentative meeting participant details. This yields in a very small amount of data to be transferred, two messages containing initiator name, start time, end time, and date (around 20 bytes each). Cancel meeting also takes around 20 bytes of data transfer. Just like set up meeting scenario, we present response time of components for cancel meeting and reschedule meeting in Figs. 11 and 12.

## 8. Related Work

Our literature survey broadly spans two areas: middleware systems for collaborative applications and the ease of developing and deploying collaborative applications, in particular calendar and travel applications. In this section, we describe related work in these two areas.

### 8.1. Middleware Systems

Here, we compare our work with other mobile middleware platforms supporting collaborative application development. Although, there is an abundant body of research carried out in middleware area in general, we review the ones that aim at supporting colloborative application development on mobile devices. Generally, mobile middleware systems can be classified as: (i) P2Pprotocol oriented systems (Fok et al., 2004; Kortuem, 2002; Mascolo et al., 2001; Kotilainen et al., 2005), (ii) dynamic distributed applications (e.g. JXTA) or IP-based client-server applications (e.g. Jini, Microsoft .NET, and others), and, (iii) middleware infrastructures supporting collaborative application development (Kortuem, 2002; Krebs et al.,2003; Cugolaand Picco,2002; Kirdaet al.,2002; Yamin et al., 2002; Gupta et al., 2009).

As elaborated in Section 1, the current technology for the development of collaborative web applications over a set of wired or wireless devices has several limitations. A few existing middleware systems have addressed the stated requirements in a piecemeal fashion. For example, Proem (Kortuem, 2002) is one such platform for developing and deploying peer-to-peer (P2P) collaborative applications in a mobile ad-hoc networking environment. Similiarly, Kotilainen et al. (2005) also propose Mobile Chedar, a peer-to-peer middleware for mobile devices using Bluetooth technology. LIME (Fok et al., 2004) is a P2P-Protocol oriented coordination model for ad-hoc networks. Commercial products such as .NET compact framework (Neable, 2002) and J2ME are also popular.

Juszczyk and Dustdar (2008) describe RESCUE, a service oriented middleware for disaster and recover application which uses efficient P2P protocols for service advertisements and discovery. Gu et al. (2005) describe a service oriented middleware for developing context aware applications using an ontology based approach. Chakraborty et al. (2004) describe issues related to service composition in mobile environments and evaluate criteria for judging protocols that enable such composition. A distributed architecture and associated protocols for service composition in mobile environments based on factors like mobility, dynamic changing service topology and device resources are presented. The composition protocols are based on distributed brokerage mechanisms and utilize a distributed service discovery process over ad-hoc network connectivity. The DISCIPLE System (Krebs et al., 2003) also supports heterogeneous collaboration over web, including mobile devices. ISAM (Yamin et al., 2002) supports infrastructure for mobile collaborative applications using java based middleware, similar to ours. MOTION (Kirda et al., 2002) is another framework for developing collaborative applications on mobile devices. Newer developments in mobile middleware include developing a mobile middleware for social networking applications (Pietiläinen et al., 2009; Gupta 2009), which falls under the broader category of collaborative applications. Gupta et al. (2009) propose MobiSoc, a middleware for mobile social computing applications.

All the aforementioned research works provide a middleware for mobile devices targeting different features. The distinguishing aspect of our work is we propose a software engineering methodology for developing both collaborative and ad hoc applications using middleware for mobile devices. Also, features such as atomic transactions over group of web objects, constraints on mobile web objects, and ease of application development methodology that are supported in SyD middleware are simply missing in the existing middleware systems. Other limitations of current middleware systems include: restricting the usage of mobile devices to only client-side programming and are incapable of being used as servers, can be applied to only

restricted domain of applications like gaming, bidding, etc., or limited group or transaction functionalities or mobility support, as further elaborated in our earlier work (Prasad et al., 2004). SyD, on the other hand, addresses all these heterogeneous data and device problems and provides a new platform technology that makes developing applications easy and independent of data format, device type and device location.

### 8.2. Calendar Application Development

Many existing calendar system‘s main goal is setting up meetings. Apart from setting up simple meetings, SyD calendar also focuses on the logic and enforces interdependencies, if any in context of meeting. Here, we compare SyD calendar with other widely used industrial calendar systems like Novell GroupWise, Microsoft Outlook, and Lotus Notes.

The Novell GroupWise offers the full range of workgroup functions, messaging, calendaring, scheduling, task management, document management, document imaging and editing, and web publishing (Novell, 2003). SyD calendar targets only at the calendar workgroup function with many inter-dependency options and automated rescheduling in case of cancel meetings. GroupWise leverages from pre-existing user profiles. The SyD directory service for a wired device needs the account information to be published only once. SyD emphasizing on the mobile work group functionality needs the publishing of proxy information for each account on to the directory to handle disconnectivity. The existing account information is synchronized with proxy information when the original account is restored from disconnections.

The Lotus Notes directory is not smart because public address book is a single-application directory, unlike GroupWise. Thus user accounts for these workgroup products are created and managed separately from existing Network Operating System accounts—requiring double the time, effort, and cost to create. A Notes application can however be rapidly built and deployed than a GroupWise application (Lotus, 1999). Outlook calendars are stored centrally and allow sharing calendar with other account’s calendar (Outlook, 2001).

When a calendar is shared, it is visible to everyone or select individuals. Outlook calendar keeps the privacy if we choose not to share our calendar details, all others can see only the slots available. Outlook allows meeting comments like conference room, details, etc., and synchronization with PDA calendar. The Exchange server 2003 provides Exchange ActiveSync for windows mobile-based devices (Morimoto et al., 2003). When set-up meeting is initiated, Exchange sends a message inviting all of the attendees. The participants can agree to attend, tentatively agree to the time, or decline. If they agree, the entry is marked in their calendar and the initiator is notified both by email and in a tracking function for that event. Microsoft Exchange

server software enables Outlook's collaborative groupware features. The Exchange Server requires lot of space and the installation procedure is complex, requires licensing and costly (Morimoto et al., 2003). The SyD middleware is very light, easy to install and part of the serving capability is hosted by mobile devices themselves.

The centralized storage of outlook calendars raises fault tolerance issues. SyD calendars are stored in a distributed fashion. When set up meeting is initiated on SyD calendars, the initiator first gets the available times from all participants. The initiator then picks a time slot for meeting and blocks that particular slot. The participants receive a meeting notification along with other meeting details in a simple text file format. A meeting can be set tentatively with participants who are already booked for the timing initiator has planned on. This provides an easy way to automate rescheduling tentative meetings when the current meetings are cancelled. Both SyD and outlook have the provision for priority of meetings. The other group logic in SyD incorporates OR, AND, and XOR logic for the inclusion or exclusion of various participants.

### 8.3. Travel Application Development

There is a lot of effort involved to induce the application logic and requires proficiency in coding and other technical details with the current state of art to compose existing web services. In the early phases of internet, the customer had to manually navigate the Internet, searching sites to organize his trip. Flights and cars had to be reserved, hotels booked, all from different websites (Oellermann, 2001). Companies like Hotwire, Priceline, Orbitz, etc., made an initial effort to transform a travel plan from multiple sites to a single website. But even now, most portals do not combine and maintain a global relation among these services (e.g. flight reservation, rental car, hotel, etc.). There is huge effort when changes are made to one aspect of an itinerary (e.g., a cancelled flight) resulting on manual changes to other sequence of events in the planned trip (e.g., canceling the car, changing hotel room reservation, etc.). Monolithic applications take a great deal of time and resources to create. They are often tied to a specific platform or to specific technologies, and they cannot be easily extended and/or enhanced. There is no effortless way to access information or perform a task without working through the graphical user interface, which can be cumbersome over a slow connection or unworkable on a portable device like a cell phone. SyD travel application provides units of application logic that can be reused between one application and another. Non-technical users will be able to easily and rapidly compose and link existing web services to create ad-hoc applications using SyD travel (Hariharan et al., 2004).

## 9. Conclusions and Future Work

We have described the high-level programming and deployment methodology of System on Mobile Devices (SyD) middle-ware which supports an efficient collaborative application development environment for deployment on a collection of mobile devices. One of the main advantages of SyD is a modular architecture that hides inherent heterogeneity among devices, data stores, and networks by presenting a uniform and persistent object view of mobile server applications and data-stores interacting through XML/SOAP requests and responses.

The paper has demonstrated the systematic and streamlined application development and deployment capability of SyD for collaborative applications composed over mobile web objects. We illustrated this design process using two application case studies: (i) a calendar of meetings application representing a collaborative application, and (ii) a travel application which is an ad-hoc collaborative application. We also presented implementation details and performance metrics for the calendar of meetings application. Specifically, we measured the bandwidth required, the storage requirements, and the response timings. The results we obtained show that the application scales well as we increase the group size and fits well within the framework of mobile devices. Therefore, SyD objects, their interactions, and the underlying techniques discussed in this paper provide a direct benefit to web services and their compositions and coordination.

In future, we would like to design a secure platform for SyD applications for different domains of applications such as social networking, emergency, disaster, and recovery applications. We also would like to expand the current architecture to include cloud based applications by integrating our rapid application development with generic interface for pluggable, web service based applications.

## References

Balasooriya J, Prasad SK. Toward fundamental primitives and infrastructure enhancements for distributed web object coordination and workflows. In: Proceedings of the IEEE international conference on web services, Orlando, July, 2005.

Chakraborty D, Joshi A, Finin T, Yesha Y. Service composition for mobile environments. Journal on Mobile Networking and Applications February 2004 [special issue on mobile services].

Cugola G, Picco GP. Peer-to-peer for collaborative applications. In: Proceedings of the International Conference on Distributed Computing Systems Workshops (ICDCSW), July 2002.

Edwards WK, Newman MW, Sedivy J, Smith T, Izadi S.Recombinant computing and speakeasy

approach. In: Proceedings of the MobiCom, Atlanta, September 2002. p. 279–86.

Fok C, Roman G, Hackmann G. A lightweight coordination middleware for mobile computing. In: Proceedings of the 6th international conference on coordination models and languages, Italy, February 2004. p. 135–51.

Fowler M, Scott K. UML distilled: a brief guide to the standard object modeling language, 2nd ed. Addison-Wesley Publication; 2002.

Garbinato B, Rupp P. From ad hoc networks to ad hoc applications. ERCIM News Journal 2003. July.

Gu T, Pung HK, Zhang DQ. A service-oriented middleware for building context-aware services. Journal of Network and Computer Applications 2005;28(1):1–18.

Gupta A, Kalra A, Boston D, Borcea C. MobiSoC: a middleware for mobile social computing applications. ACM/Springer Mobile Networks and Applications Journal (MONET) 2009;14(1):35–52.

Hariharan A, Prasad SK, Bourgeois AG, Dogdu E, Navathe S, Sunderraman R, et al.. A framework for constraint-based collaborative web service applications and a travel application case study. In: Proceedings of the international symposium on web services and applications, 2004. p. 866–72.

Joshi J. A system for rapid configuration of distributed workflows over web services and their handheld-based coordination. Master's thesis, Georgia State University, 2005.

Kirda E, Fenkam P, Reif G, Gall H. A service architecture for mobile teamwork. In: Proceedings of the 14th international conference on software engineering and knowledge engineering, 2002.

Kotilainen N, Weber M, Vapa M, Vuori J. Mobile chedar—a peer-to-peer middleware for mobile devices. In: Proceedings of the third IEEE international conference on Pervasive Computing and Communications Workshops (PERCOMW'05), 2005. p. 86–90.

Kortuem G. Proem: a middleware platform for mobile peer-to-peer computing. ACM SIGMOBILE Mobile Computing and Communications Review (MC2R) 2002;6(4). October.

Krebs AM, Ionescu MF, Dorohonceanu B, Marsic I. The DISCIPLE system for collaboration over the heterogeneous web. In: Hawaii International Conference on Computer System Sciences, 2003.

Krone O, Chantemargue F, Dagaeff T, Schumacher M, Hirsbrunner B. Coordinating autonomous entities. The Applied Computing Review, [special issue on coordination models languages and applications] 1998.

Lotus. Moving from novell groupwise to lotus domino R5. IBM Redbooks Publication; 1999.

Juszczyk Lukasz, Dustdar Schahram. A middleware for service-oriented communication in mobile disaster response environments. In: Proceedings of the 6th international workshop on middleware for pervasive and ad-hoc computing, Leuven, Belgium, 2008. p. 37–42.

Mascolo C, Capra L,Emmerich W. An XML-based middleware for peer-to-peer computing. In: Proceedings of the international conference on peer-to-peer computing, Linkoping, Sweden, 2001.

Morimoto R, Gardinier K, Noel M, Coca J. Microsoft exchange server unleashed, 1st ed Sams Publication; 2003.

Neable C. The NET compact framework. IEEE Pervasive Computing Magazine; 2002. October–December.

Novell. The groupwise advantage, White papers from Novell, January 2003.

Oellermann WL. Architecting web services. Apress Publication; 2001.

Outlook. Building applications with microsoft outlook version 2002. Microsoft Press Publication; 2001.

PhanT, Huang L, Dulan C. Integrating mobile wireless devices into the computational grid. In: Proceedings of the MobiCom, Atlanta, September 2002. p. 271–8.

Pietiläinen A, Oliver E, LeBrun J, Varghese G,Diot C. MobiClique: middleware for mobile social networking. In: Proceedings of the 2nd ACM Workshop on Online Social Networks (WOSN ’09), Barcelona, Spain, 2009. p. 49–54.

Prasad SK, et al.. System on Mobile Devices (SyD): kernel design and implementation. In: Proceedings of the international conference on mobile systems, applications, and services, poster and demo presentation, San Francisco, May 5–8, 2003a.

Prasad SK, et al.. Implementation of a calendar application based on SyD coordination links. In: Proceedings of the 3rd international workshop on internet computing and E-commerce in conjunction with the 17th annual International Parallel & Distributed Processing Symposium (IPDPS), Nice, France, April 22–26, 2003b.

Prasad SK,Balasooriya J. Web coordination bonds: a simple enhancement to web services infrastructure for effective collaboration. In: Proceedings of the 37th Hawaii International Conference on System Sciences (HICSS-37), Hawaii, January 2004.

Prasad SK, et al.. System on mobile devices (SyD): a middleware testbed for collaborative applications over small heterogeneous devices and data stores. In: Proceedings of the ACM/IFIP/USENIX 5th international middleware conference, Canada, October 2004.

Prasad, SK, Balasooriya, J. Fundamental capabilities of web coordination bonds: modeling Petri

nets and expressing workflow and communication patterns over web services. In: Proceedings of the Hawaii International Conference on System Sciences (HICSS-38), Hawaii, January 2005.

Prasad SK, Bourgeois AG, Madiraju P, Malladi S,Balasooriya.J. A methodology for engineering collaborative applications over mobile web objects using SyD middleware. In: Proceedings of the 2005 IEEE International Conference on Web Services (ICWS 2005), Orlando, July 2005.

Pressman RS. Software engineering: a practitioner's approach, 4th ed New York: McGraw-Hill; 1997.

Yamin A, Augustin I, Barbosa J, Silva J, Geyer C,Cavalheiro G. Collaborative multilevel adaptation in distributed mobile applications. In: Proceedings of the international conference of the Chilean Computer Science Society (SCCC), November 2002.

## Appendix

### Figure 1: SyD Architecture

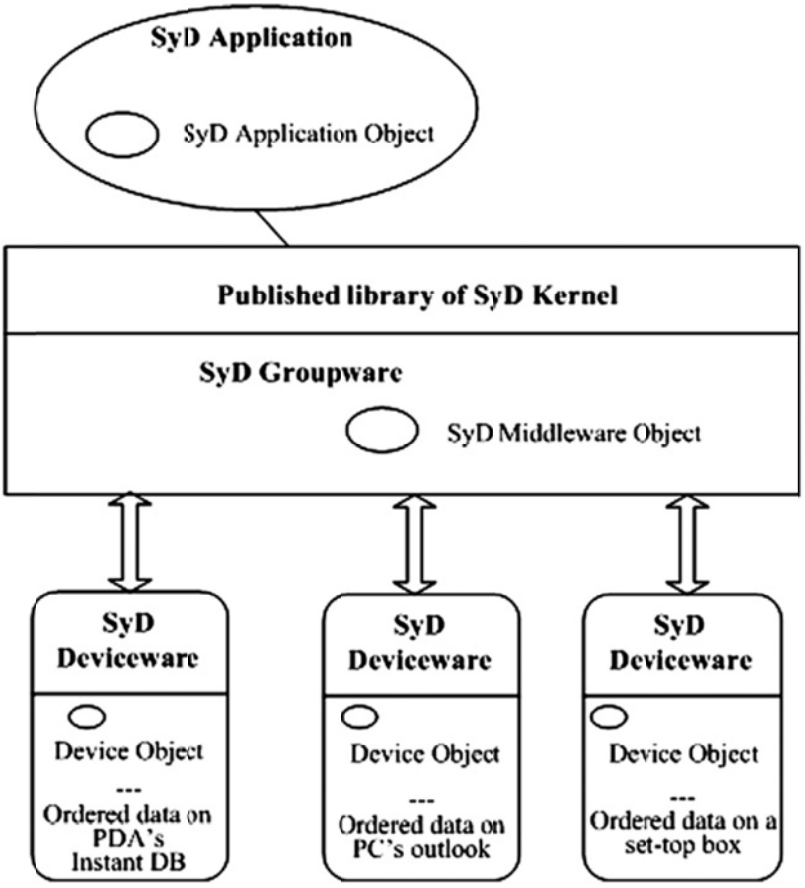

**Figure 2: Interaction among Modules of SyD Kernel**

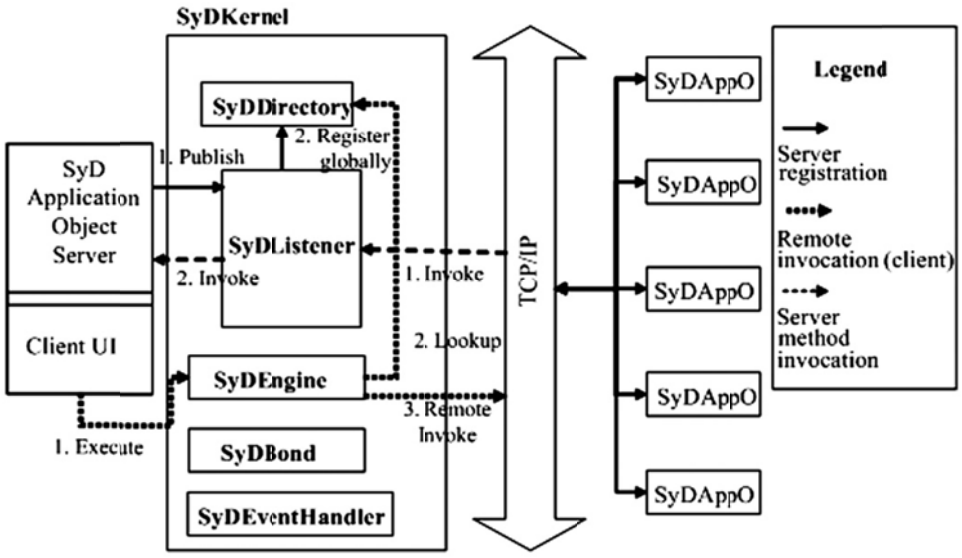

**Figure 3: A Scheduled Meeting**

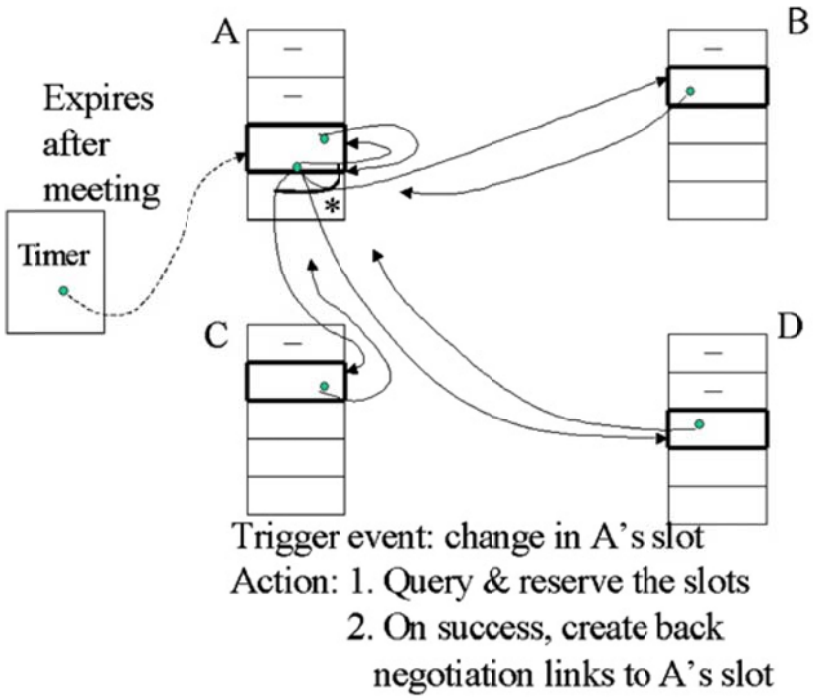

**Figure 4: A Tentative Meeting**

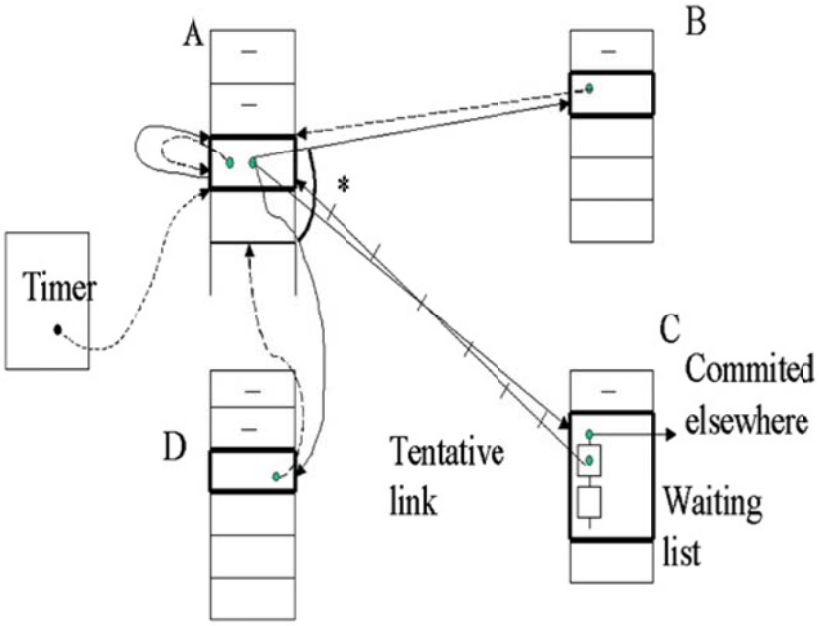

**Figure 5: A Collaborative Application Design Process**

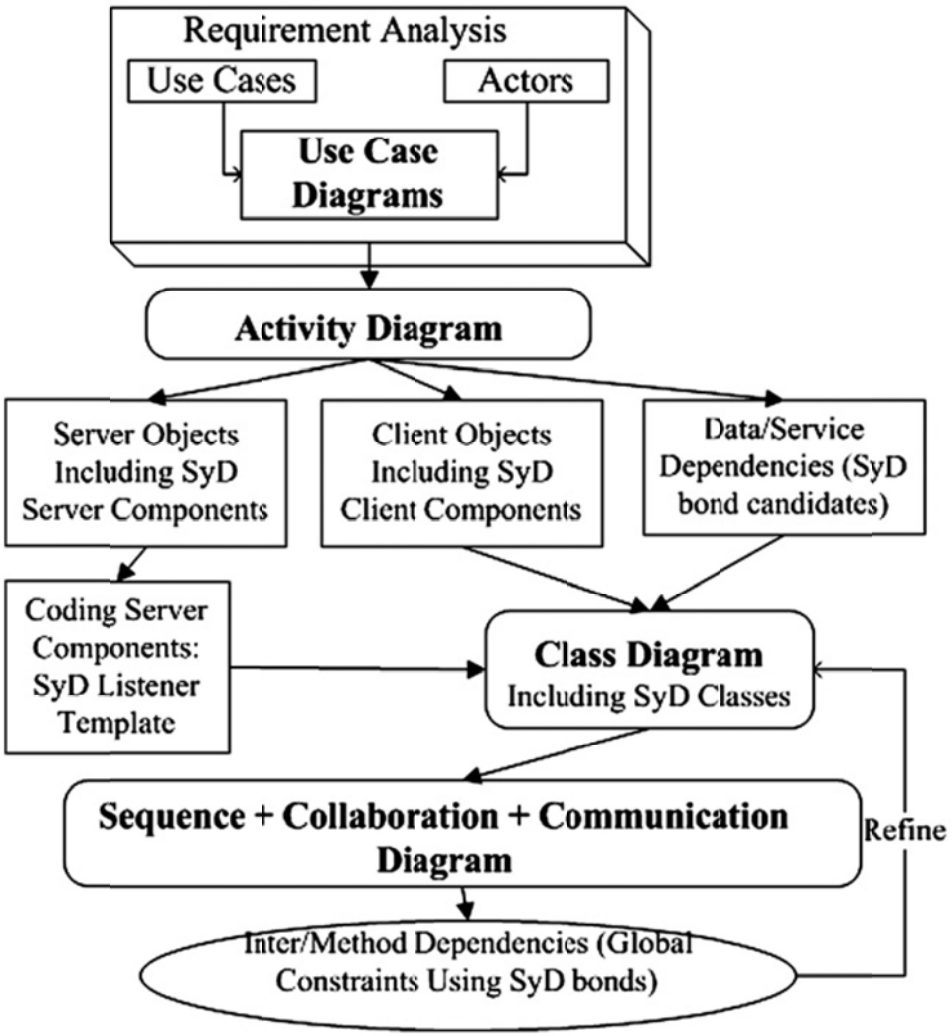

**Figure 6: CANCEL MEETING Activity Diagram**

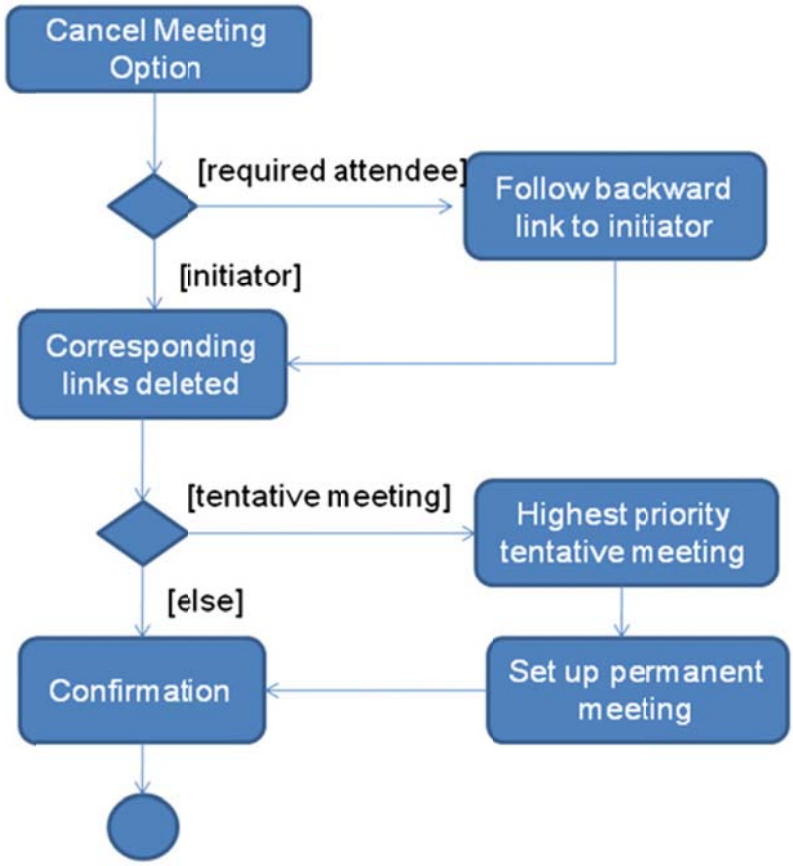

**Figure 7: Automatic Triggers of Methods**

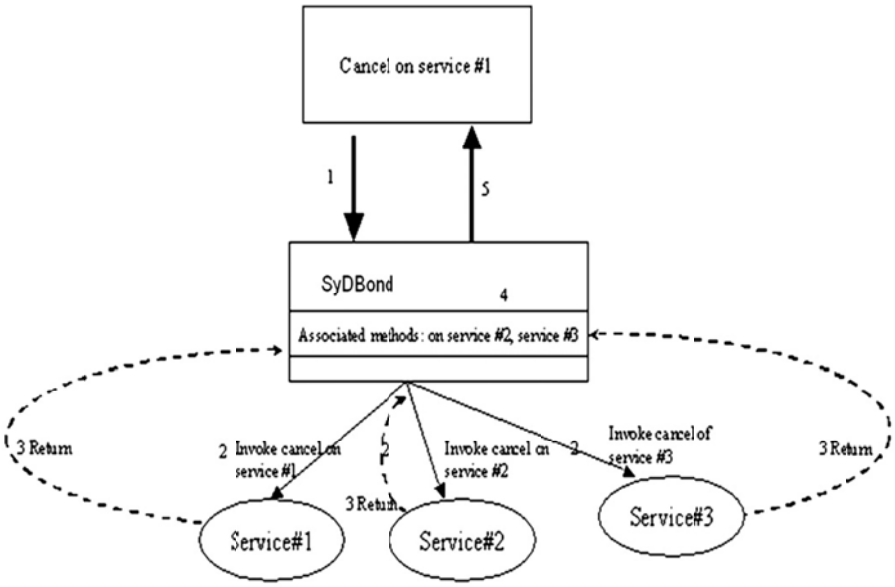

**Figure 8: SyDBond as Centralized Coordinator – Travel Reservation Application**

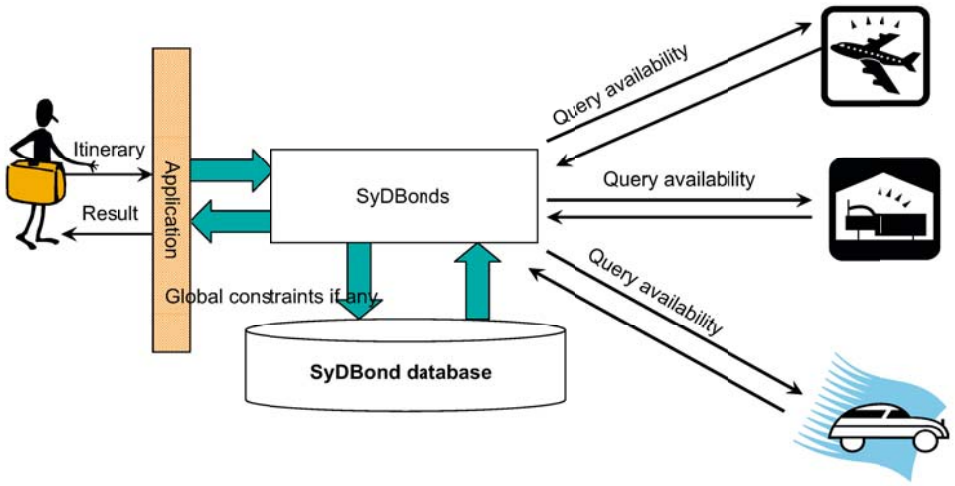

**Figure 9: Response Time for Three Scenarios**

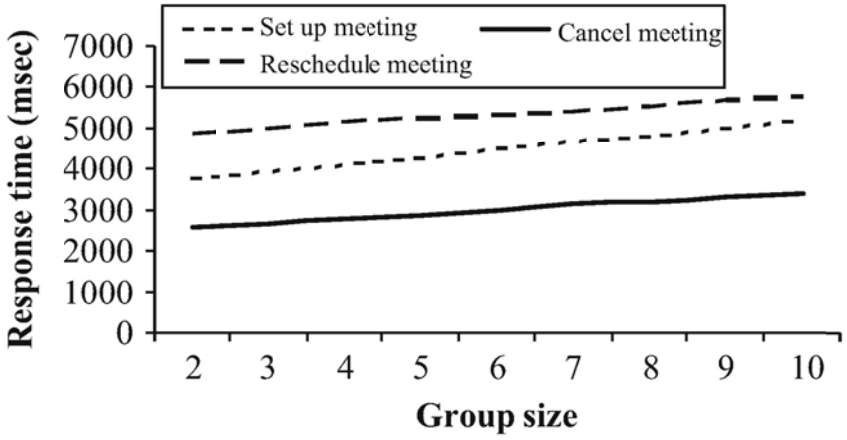

**Figure 10: Set Up Meeting Response Time for Components**

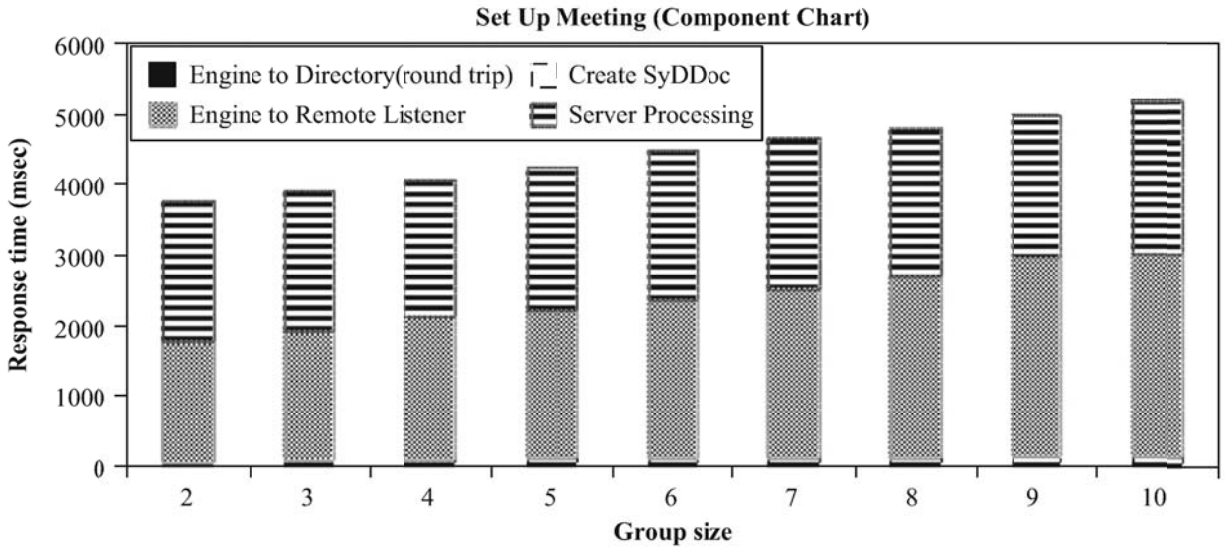

**Figure 11: Reschedule Meeting Response Time for Components**

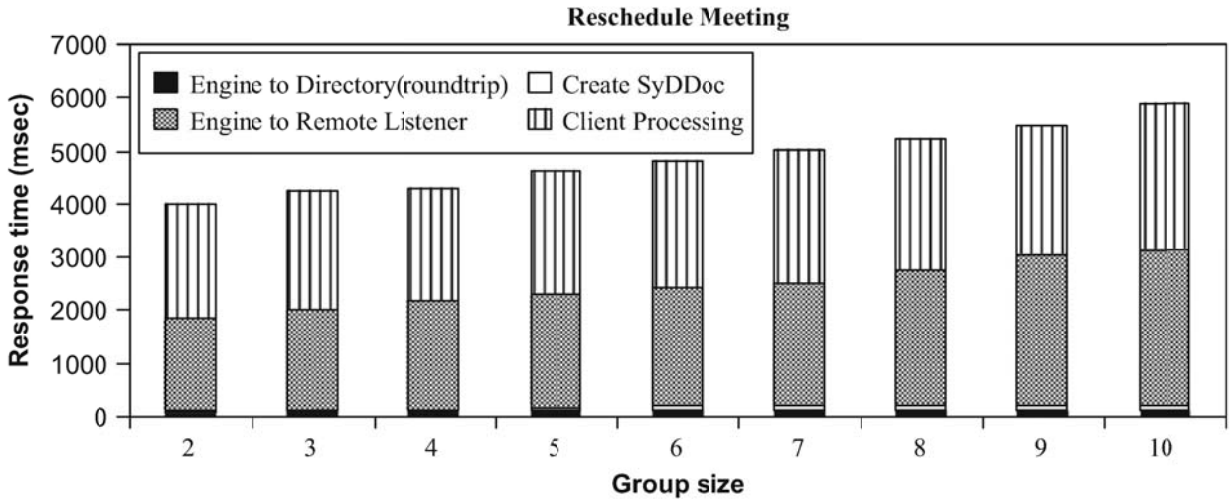

**Figure 12: Cancel Meeting Response Time for Components**

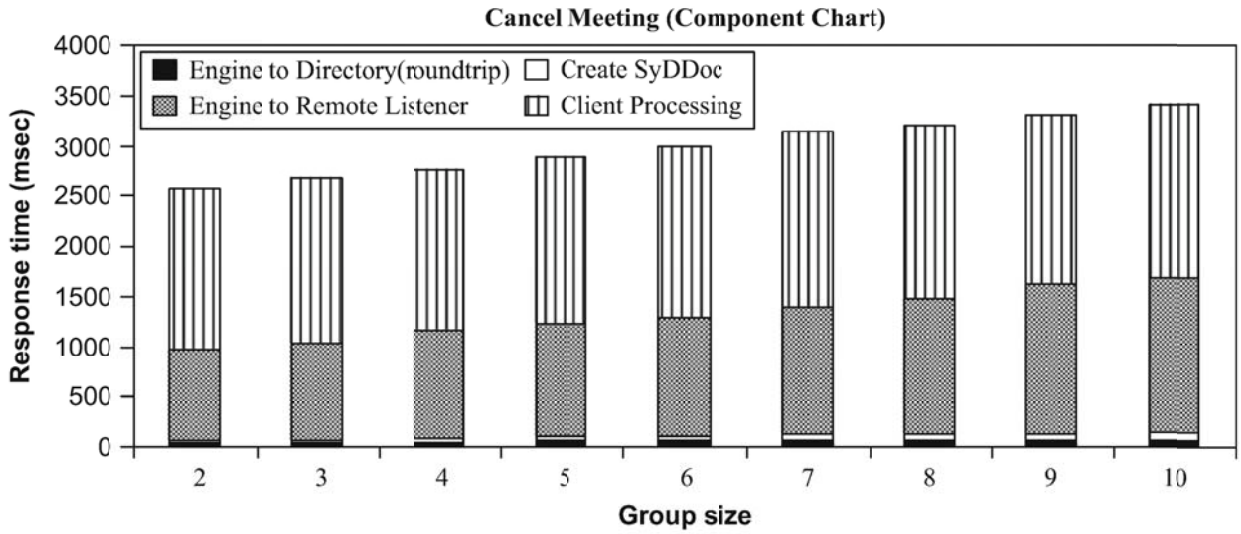

**Table 1: CANCEL_MEETING Use Case**

| **Use case name** | CANCEL_meeting |
|---|---|
| *Participating actors* | Application, initiator, system |
| *Entry condition* | 1. Cancel meeting option is selected by the Initiator or is invoked by system. |
| *Flow of events* | 2. System invokes CANCEL_MEETING.<br>3. Confirmation of cancel meeting sent to all attendees.<br>4. System checks for any associations waiting on the initiator.<br>6. All the associations waiting up on are now converted to confirmed status.<br>7. All the associations are informed of the change. |
| *Exit condition* | 8. Return to main menu. |
| *Special requirements* | |