RCal: An Autonomous Agent for Intelligent Distributed Meeting Scheduling

Rahul Singh
kingtiny@cs.cmu.edu

CMU-RI-TR-03-46

Submitted in partial fulfillment of the requirements for the degree of
Master of Science in Robotics

The Robotics Institute
Carnegie Mellon University
5000, Forbes Ave
Pittsburgh PA 15213

November 2003

Keywords: Intelligent agents, Distributed Systems, Distributed Meeting Scheduling, Artificial Intelligence, Contract Net Protocol, Automated Negotiation, Semantic Web

Abstract

Meeting scheduling is an inherently difficult and time consuming task that requires multiple parties to interact and consult their various calendars in order to agree upon a common time. These interactions must take into account numerous constraints that are set by the participants and there is no a-priori knowledge of all the possible constraints that may apply when scheduling a meeting. In addition to this there is an overhead in terms of time and effort involved in scheduling a meeting that can sometime make it impossible to schedule an event, especially if the number of attendees is large.

This report details the design and implementation of an intelligent meeting scheduling agent that assists humans in office environments by scheduling meetings. RCal agents negotiate with each other on the behalf of their users and agree on a common meeting time that is acceptable to all the users and abides by all the constraints set by all the attendees.

RCal is a distributed problem solver where each RCal agent forms a component of the problem solving engine by proving its own inputs as the problem converges towards a solution. In addition to this RCal is a robust, user-friendly system that integrates well with the latest computing environments and works efficiently to aid the human user in the task of scheduling meetings.

Thesis Supervisor: Dr. Katia Sycara
Research Professor, Robotics Institute

Acknowledgements

The work presented in this report represents the contributions of numerous people whose support made this work possible. First and foremost I would like to thank Prof. Katia Sycara for being my advisor and mentor and for supporting me in this project. I am also indebted to her for introducing me to the area of multi agent systems and distributed artificial intelligence and for the opportunities for pursuing cutting edge research in her research group at CMU. I am especially struck by her enthusiasm for building real systems and the emphasis given to intelligent systems that operate in the real world in open environments.

I would also like to thank Joseph Giampapa for his continued support and feedback through out this project and especially when I was working out the bugs in my code. I am also grateful for his additional support for me in all other areas and while working with him on project teams. Thanks also goes to Terry Payne for his contributions towards integration with the semantic web and for the designing an excellent user interface. I would like to thank Sean Owens for helping me find and remove some especially nasty bugs and for his support, friendship and patience.

Last but not least, I would like to thank my parents for their complete support in everything I have pursued.

Table of Contents

List of Figures

1. Introduction

Scheduling meetings and events is common task accomplished in everyday life where an individual or organization hosts an event and invites other individuals to attend. Meetings - either formally established or casual agreements over the phone – are scheduled by a process that finds a time that is mutually acceptable to the host as well as to all the attendees. The nature and type of the constraints set by the participants may be explicit – *existing meetings make it impossible to meet at certain times*; implicit – *I prefer to avoid meetings before 10am*; cultural – *social events should be restricted to weekends*; or any combination thereof.

Scheduling a meeting can be a lengthy and time consuming process that involves contacting the attendees, soliciting free times from them, finding a free time slot that is acceptable to all the parties and then notifying the attendees once an appropriate time is found. The length of time consumed in scheduling a meeting is unknown and may vary from a few minutes to days. This is the overhead involved with scheduling a meeting that increases with the number of attendees and other resources that must be coordinated for the event to occur. Tools that assist in the process of meeting scheduling can reduce these overhead costs and relieve humans of the burden associated with this process.

Personal Information Managers (PIM) have become increasingly common in recent years and most people use them to maintain a record of their current and upcoming meetings and events. Improvements in software have led to the development of services that provide some assistance in scheduling meetings. A commonly found feature in the PIMs available today is that of publicly publishing calendars on a shared server.

People wishing to organize meetings can use this shared information to determine whether an appropriate meeting time is available. This shared view of the various information sources available at one place allows for the easy selection of an appropriate meeting time that is acceptable to all the attendees. It is then a simple matter for the host of the meeting to notify the attendees of this meeting time so that they may mark their calendars.

While this approach seems feasible it has the following drawbacks.

1. The process of selecting an appropriate meeting time is a manual and time-consuming one. The host may have to scan through numerous calendars and across large date ranges to find an appropriate time.
2. The privacy implications of such an approach make it inflexible to publishing private data that involves the whereabouts of an individual.
3. The publish-and-subscribe mechanism employed may allow only a small subset of users to access the system. Users in an organization may receive access to the system only if they use a particular PIM and its associated server.

Automated meeting scheduling can reduce or eliminate all of the drawbacks above and involves applying techniques of distributed problem solving and automated negotiation where agents negotiate with each other on the behalf of their users. Meeting scheduling is solved by a protocol based negotiation where the problem of finding a mutually agreeable time is solved in a distributed manner by contributions made by multiple parties accessing various information sources.

The meeting scheduling problem can be represented as a set of distributed nodes and knowledge sources that negotiate with each other to reach a consensus on allocating a resource given a set of constraints. The resource to be allocated is time, the distributed information sources are the various calendars of the attendees and the constraints are numerous and unrelated – every attendee wishes to meet at times dictated by issues such as availability, convenience, social issues such as holidays, festivals, relative importance of the subject of the event and unfounded reasons – *I just don't feel like meeting*. The distributed nodes – the attendees or their agents – reason about information gathered from the information sources – their calendars – and contribute towards solving the problem.

The RETSINA Calendar Agent (RCal[1]) built on the RETSINA AI infrastructure [25] is an agent that augments a widely used Personal Information Manager (PIM) – Microsoft Outlook™ [20]. RCal combines knowledge about its user's current schedule, information about colleagues and friends (using MS Outlook 2000™'s Contact entries) and knowledge garnered from the Semantic Web [10] to better automate meeting scheduling and management. In this scenario every user running MS Outlook™ on his or her desktop also has an instance of RCal (figure 1) running in the background acting on their behalf.

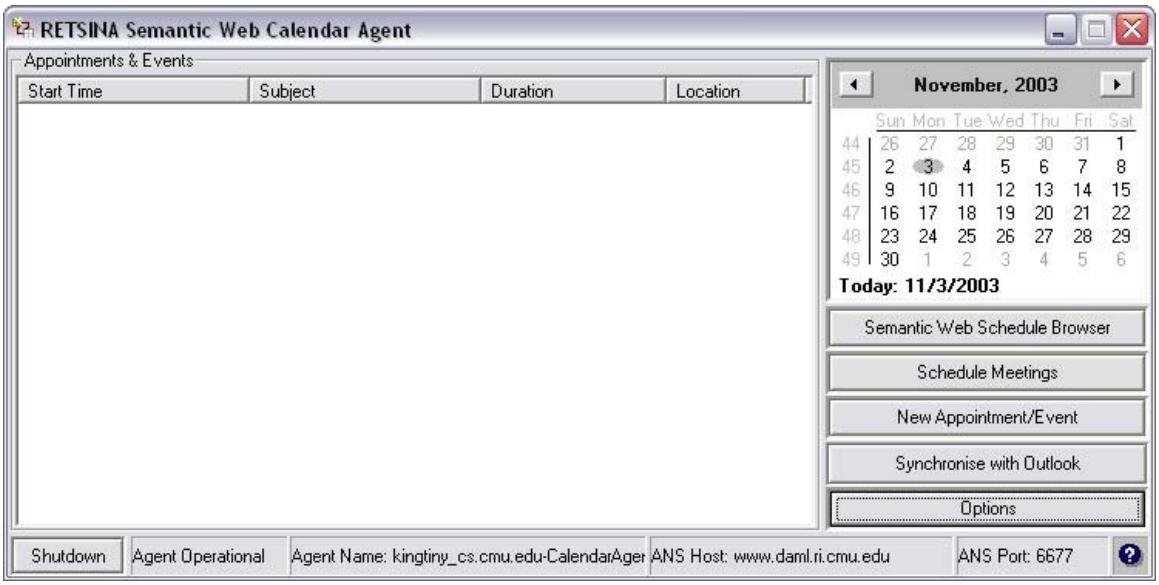


Figure 1: RCal Main Window

[1] RCal is available for download from http://www.cs.cmu.edu/~softagents/Cal

RCal (figure 1) schedules meetings for its user, updates the user's calendar with schedules from the Semantic Web [10], interacts with Web Services that may provide additional relevant information pertaining to scheduled meetings, and provides alerts based on occurring events.

When scheduling meetings on behalf of its user, RCal negotiates with its peers to find mutually agreeable times based on the user's availability and preferences. RCal supports two types of distributed meeting negotiation:

1. Multi-party negotiation involves the host of the meeting soliciting available meeting times from the attendees and then evaluating all the returned time slots for an applicable time. This is the main mode of negotiation and allows for multiple rounds of negotiation until a time is found that is appropriate to all attendees involved.

2. Appointment-Request negotiation involves a process or person requesting an appointment from RCal. RCal processes the request and examines the user's calendar before returning an affirmative or a negative response. In the former case the appointment is granted and scheduled while the latter case implies that the host is busy at the specified time. RCal may return an alternative time in the latter case.

RCal's multi-party negotiation occurs when someone desires a meeting with one or more individuals, each of which employ their own RCal agents to manage their schedules. RCal goes through several rounds of automated negotiation with other RCal agents until all the agents agree upon a common time. Users can instruct RCal to schedule a meeting by specifying a particular time they would prefer or a window within which an appropriate slot of the specified duration should be found. For example, a user may request *"an hour's meeting with all the members of the project group within the next two days"*. The negotiation uses the Contract Net Protocol [13], which starts with an initial document (a contract) specifying details of the event such as its attributes (subject, body location etc) and its constraints (start time, end time and duration). The contract is broadcast to the other RCal agents, which consult their user's calendars and reply with one of three types of bid:

- Accept the contract for the meeting at the specified time;
- Reject the contract outright (i.e. do not wish to meet);
- Reject the time specified by the contract and propose alternative time slots.

The host evaluates all bids upon receipt and looks for a time that is acceptable to all the attendees. If a common time is found the host sends out awards corresponding to the received bids and the RCal agents update their respective calendars before sending out confirmations that terminate the negotiation. If no common time can be found, the negotiation restarts with a new contract generated within the original constraints specified

by the host. This iteration continues until a meeting time is identified or no new contract can be generated due to the original constraints set by the host of the meeting.
Figure 2 shows the interactions between the RCal agents and the MS Outlook 2000™ clients they consult.

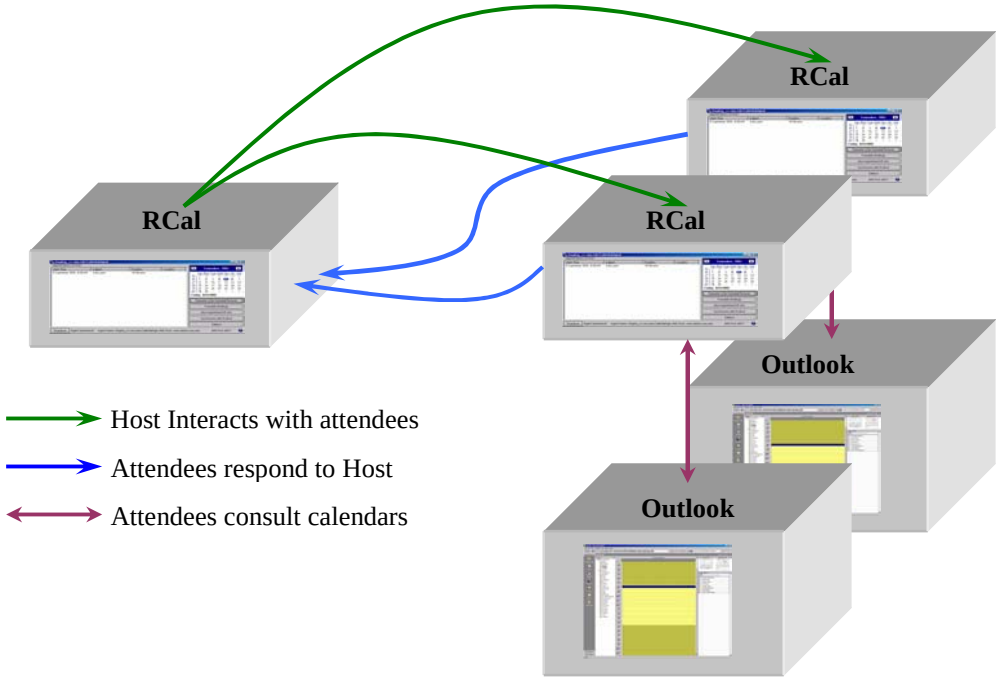


Figure 2: RCal agents negotiate with each other to determine an appropriate meeting time

This report describes the general problem of meeting scheduling as it applies to office and work environments and presents one design and implementation to solve this problem in a production environment. Details of the implementation of RCal include design decisions and tradeoffs that were made as well as concrete implementation details for the system developed.

Related Work

Distributed meeting scheduling is an open problem in the area of research in Artificial Intelligence with numerous techniques introduced with the aim of finding a complete, optimal and sound solution to the problem. Research in automated negotiation has produced a number of successful candidates but there has been no system built that solves the problem in a general and robust way in an open environment.

Commercial calendar management solutions [15,17,20,21] are simple systems that provide facilities for the management of the information associated with scheduling meetings while leaving the task of finding a meeting time to the user. Such systems are typically server based and offer a very specific configuration that can be difficult to integrate into the existing environment. The basic tenet of a commercial server based system requires the sharing of calendars where every user in the organization or managed domain makes his or her calendar available. Such systems do little to ease the cognitive load on the user.

Recent advances in research in the fields of computer science and artificial intelligence have produced some solutions that look promising. Garrido and Sycara [2,3] present a decentralized solution where autonomous agents negotiate amongst themselves for various parameters towards determining a meeting time. Based on a negotiation protocol presented in [3], the system takes into account each agent's preferences, calendar data and the attributes of the meeting to be scheduled. Shintani, Ito and Sycara [8] present another technique based on automated negotiation that relies on a method of persuasion that aims to maximize the expected payoffs of an agent involved in scheduling a meeting.

Garrido, Brena and Sycara [9] also present a method of meeting scheduling based on maintaining a cognitive model of each agent in the system. Each agent maintains a set of parameters that together form a model of the actor in the scheduling process. Using this belief set an agent can make predictions of behavior of its peers and along with its own intentions and desires can execute appropriate actions. The system aims to build a social environment consisting of the attendees of a meeting where each agent adapts to the environment formed.

Sen and Durfee [4,5,6] present a distributed meeting scheduling system based on contracting [13] and explore various parameters of the protocol for its contribution to the efficiency of the process. They also present a prototype that uses email as a communication medium and present some preliminary results. The technique presented does not take user preferences into account but further work by Sen, Haynes and Arora [7] presents a voting mechanism for managing user preferences while scheduling meetings.

There has also been some work in managing user preferences and meeting scheduling. Mitchell et. al. [14] present a machine learning system that learns the user's preferences associated with meeting scheduling. While the social and distributed aspects of the

problem are not addressed, the solution presented along with the formulation the problem is a significant contribution to this area of research.

Another recent work in distributed meeting scheduling is by Ephrati, Zlotkin and Rosenchein [1]. They present three scheduling mechanisms for closed systems using an economics-based point system that allows each user to specify their preferences. There is no attempt at exploring meeting scheduling in open systems.

In the area of calendar management and workgroup oriented systems, there are numerous commercial software products that provide facilities for managing calendar data. Some of these such as Microsoft Outlook [20] are client based while others are more workgroup and server oriented. In recent years there has also been the proliferation of web based calendar management system such as those provided by Yahoo! [18] and MSN [19]. New technologies such as XML web services are also giving rise to calendaring, messaging and reminder services such as those provided MSN Alerts and Yahoo! Alerts.

There are also new standards being developed to support software products and tools that deal with calendar data. The iCalendar specification [22] provides a data format for representing calendar and event data for transmission across networks. It is widely used by systems such as Outlook [20] and Apple iCal [21]. Work in the area of the Semantic Web [10] is also producing ontologies to represent and reason about time [23] and calendar data [24].

2. RCal Architecture

RCal is a distributed problem solver – an intelligent agent that interacts with its peers to schedule meetings on the behalf of its human users. This is a peer-to-peer protocol driven system that uses a slightly modified version of the Contract Net protocol [13] for distributed meeting scheduling.

Solving the meeting scheduling problem involves integrating information from a set of distributed, loosely coupled knowledge sources - the calendars of the various attendees - in order to make a decision that allocates a shared resource – time – amongst the attendees in a cooperative manner. The nature of the problem lends it well to a solution based on the Contract Net Protocol.

The Contract Net protocol is a task allocation algorithm. It aims to solve the general problem of selecting one of a number of possible candidates to perform a given task given a set of constraints. In this mechanism the host broadcasts a message to a set of candidates outlining the task that must be carried out. The candidates then evaluate the task and reply with bids outlining their capabilities and their applicability to the task announced. The host can then use this information to select a bidder to assign the task.

In a similar fashion, the agent wishing to schedule a meeting plays the role of the host and solicits meeting times from the attendees who play the role of bidders in the negotiation. The meeting is announced using a contract that specifies the constraints of the meeting such as the earliest and latest meeting times and the duration. The attendees evaluate the contract and return bids that specify times when they are available to meet.

The host evaluates all bids and searches for a common meeting time. This process continues until a suitable meeting time is found or it is determined that no mutually acceptable meeting times are available.

3.1 High Level Architecture

The RETSINA Calendar Agent (RCal) consists of the following major components:

1. The Calendar Database
2. The Distributed Meeting Scheduling Engine
3. Communication Infrastructure

Every RCal agent interacts with its peers via the RETSINA agent communication infrastructure that supports the system. Each agent acts on the behalf of its human user and as such is a running process that runs in parallel with a PIM that acts as a persistent Calendar Database. RCal uses Microsoft Outlook for its persistent storage of calendar data. The reasons for this choice are twofold. Firstly Outlook is a popular PIM that is used to maintain calendars thus increasing the adoption of the system by users already using Outlook. Secondly Outlook provides the necessary data structures and API for storing, retrieving and displaying calendar data. RCal uses Outlook only as a persistent

store and maintains its own in-memory primary calendar database for meeting scheduling because (a) the interaction with Outlook via COM automation is relatively slow and (b) Outlook does not provide any facilities for searching the database and making complex queries. Section 4 gives the design and implementation details of this component.

The communication layer is built using the RETSINA [25] agent API that provides access to the RETSINA communicator [26], Agent Name Server (ANS) client, and Belief DB and Logging modules. The Distributed meeting scheduling engine sits between the communication layer and the data store and performs tasks related to scheduling meetings. It is divided into the following modules:

1. Contracting Engine
2. Bidding Engine
3. Scheduling Engine

Figure 3 shows the architecture diagram of the various components of RCal and their role in the overall system. Section 5 gives the details of the Distributed Meeting Scheduling engine while section 6 examines the communications infrastructure.

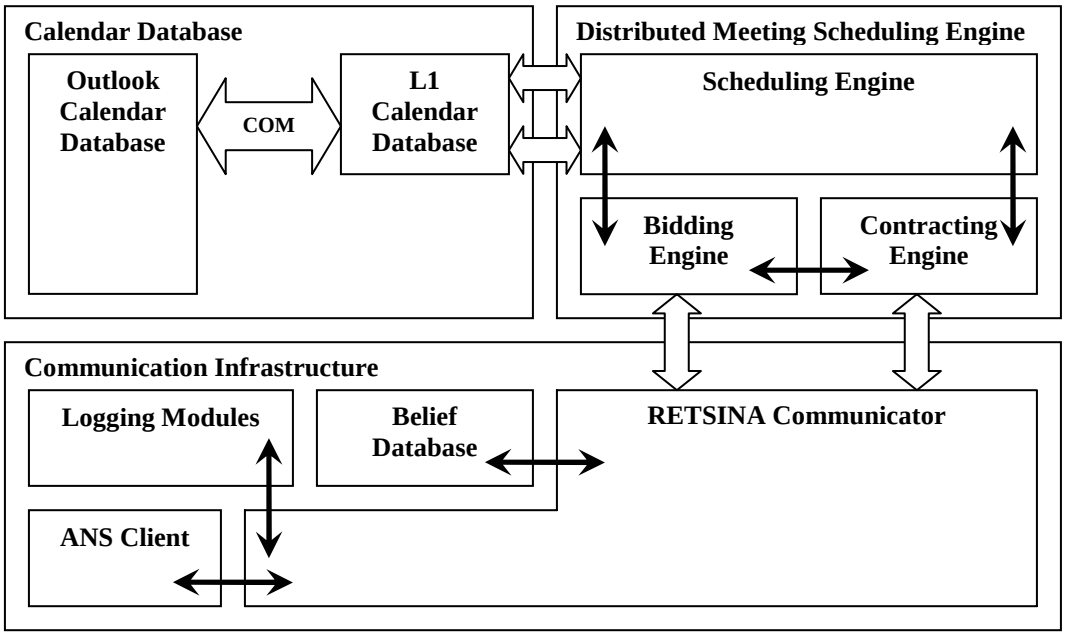


Figure 3: RCal Modular Block Diagram

3.2 Client Server vs. P2P architecture

RCal is built on a Peer-to-Peer architecture where each RCal agent interacts directly with its peers on the network. This design offers the flexibility and redundancy that comes with P2P systems. Client-Server architectures on the other hand are well suited to facilitating interactions that include numerous heterogeneous nodes that operate in

various environments some of which may not allow direct connections. The various tradeoffs that were made with respect to the design fall into four major categories:

1. *Scalability*: In P2P architectures each agent connects directly to another agent and interacts with it without the need for an intermediary. Such a system scales up well because it allows an unlimited number of distinct nodes to interact without a single point of failure that receives the entire load. The system can thus schedule multiple meetings concurrently limited only by the number of communication nodes each RCal agent can accommodate. A client/Server architecture would require a single server that would broker all the communications and would thus be a single point of failure for the system. In addition to this multiple load balancing servers would need to be added as the system grew and this would require additional synchronization overhead to keep the servers in sync with each other.

2. *Agent Naming and Discovery*: It is necessary for an agent to locate and identify its communication partner prior to any interaction. Thus each agent is given a unique name that is federated by the agent infrastructure – in this case the agent name server (ANS). This unique name is used to locate the agent, identify it and possibly authenticate it prior to any communication. A client/Server architecture is more suitable for such a situation where a single point of service must assign names to agents in the system. Achieving this in a P2P system would be more difficult due to the open nature of P2P networks and thus RCal behaves as a client that connects to a server – the ANS – in order to validate its name for uniqueness.

3. *Communication and Connectivity*: The nature of a P2P network demands that every node in the network be accessible from every other node for maximum flexibility and efficiency. In the context of meeting scheduling, the nodes that are inaccessible – due to factors such as firewalls – will not be able to participate in negotiations and no hosts will be able to schedule meetings with them. Thus all meeting requests that have these nodes as required attendees will fail. However such an always connected scenario is not possible ubiquitously the biggest reason being agents behind firewalls being uncontactable. A client/server architecture would be a step towards solving this problem because it would force all clients to actively connect with the server and then wait for meeting requests on that connection. Once meeting requests arrive then the host can either continue further interaction via the server or the server could hand off the connection to the host. RCal's P2P architecture demands always-on connectivity but allows for connection servers to be installed. In such a scenario an RCal agent behind a firewall would connect with a connection server – known a-priori – and then would wait for contract announcements on this connection. The pull nature of this interaction requires the connection with the connection server to be maintained at all times.

4. *Security and Authentication*: Privacy, authentication and general security issues are of primary importance in the domain of meeting scheduling. One of the motivating factors for an automated negotiation is to avoid publishing entire calendars – a technique that is currently used. When scheduling meetings RCal agents contact their

peers and announce contracts. Due to the P2P nature of the system, every agent would need to authenticate every other agent prior to any communication whatsoever. This adds an overhead to the negotiation which can be substantial. It also requires authentication modules and code to be integrated into every agent which requires an additional processing overhead on every platform in terms of the memory and processing power required. This can elevate to levels where it becomes prohibitive for application on low power platforms such as mobile devices. The overhead involved with a security infrastructure can be reduced by building the security infrastructure over a client/server architecture. RCal agents can authenticate themselves to an authentication server and maintain that connection which can then be used to verify the identity on subsequent calls.

There is no perfect architecture that solves all the problems and numerous tradeoffs need to be made. RCal uses a P2P architecture for its primary problem solving capabilities – meeting scheduling - and utilizes the benefits of a client server architecture where necessary.

4. Calendar Management

RCal uses a two level calendar store for efficient calendar management. The first level is a volatile in-memory store that is used to maintain an up-to-date copy of the user's calendar which is used as a reference while scheduling meetings. This internal data structure changes frequently and is used to maintain state information as well as event data of the negotiations as they are processed. This data structure provides facilities for searching events and for performing complex queries on the data in an efficient manner.

RCal also uses a second level persistent calendar data store that is used to store only scheduled event data – data for events already scheduled. RCal integrates with Microsoft Outlook via COM automation and uses the calendar database offered by Outlook as the persistent store thus utilizing the calendar display features and data file management features. In addition to this users that already use Outlook's calendar to maintain their schedules can use the meeting scheduling capabilities offered by RCal without having to move to another calendar. The Outlook Calendar serves as an up-to-date copy of the user's calendar that ensures that RCal is notified of all the events that the user is scheduled to attend.

4.1 Outlook Integration

When RCal starts up it initializes COM services (component object model services) and then loads the Microsoft Outlook objects via COM automation. This launches the outlook.exe process as a COM server and enables access to all the features of Outlook such as the calendar.

The major steps involved in integrating with Outlook are as follows:

1. *Initializing COM*: Initializing COM services is the first step in any application using COM. It is necessary to initialize COM (once and only once on each thread from which you use it) in order to access the class loader services and libraries offered by the operating system that will load the necessary COM objects that you wish to use.

   COM initialization is a simple matter of calling the COM API function `CoInitialize()` (or `CoInitializeEx()`) and checking the returned HRESULT code for success as follows

   ```
   HRESULT hr = CoInitialize(NULL);
   if(FAILED(hr))
   {
         //COM Initialization failed.
         //Handle the error and exit gracefully
         return 0;
   }
   ```

   The above call sets the application concurrency model to single threaded apartment (STA). MFC also provides an API for initializing COM that takes no parameters and

returns a BOOL that indicates success or failure. This is the recommended method of initializing COM in MFC applications.

```
If(!AfxOleInit())
{
      //Could not initialize OLE/COM.
      return 0;
}
```

`AfxOleInit()` internally calls `OleInitialize()` which in turn calls `CoinitializeEx()` and sets the concurrency model to single threaded apartment (STA). Once the above API calls succeed and COM is initialized the required COM objects can be loaded. In this case it is necessary to initialize Outlook and load the Outlook object model.

2. *Initializing Outlook*: The typical way to create an instance of a COM object is to use the `CoCreateInstance()` API. However a recommend method is to use Active Template Library (ATL) smart pointers that make the task easier. It is necessary to import the Microsoft Outlook object library (msoutl.olb) and mso.dll so that the necessary declarations can be found.

```
#import “C:\Program Files\Common Files\Microsoft
Shared\Office10\mso.dll” rename(“RGB”, “OutlookRGB”)

#import “C:\Program Files\Microsoft Office\Office10\msoutl.olb”
rename(“ExitWindows”, “OutlookExitWindows”) rename(“CopyFile”,
“OutlookCopyFile”)
```

Initializing Outlook is a simple matter of using the CreateInstance API as follows

```
Outlook::_ApplicationPtr pOutlook;
HRESULT hr;
hr = pOutlook.CreateInstance(__uuidof(Outlook::Application));
If(FAILED(hr))
{
      //Failed to initialize Outlook
}
```

Once the Outlook application object is initialized and loaded, it can be used to access other modules in the Outlook object model. In order to access the calendar data it is essential to log on to the MAPI profile as follows:

```
Outlook::_NameSpacePtr pNamespace;
pNamespace = pOutlook->GetNameSpace(“MAPI”);
COleVariant covOptional((long)DISP_E_PARAMNOTFOUND, VT_ERROR);

hr = pNamespace->Logon(covOptional, covOptional, covOptional,
covOptional);
if(FAILED(hr))
{
      //failed to logon on to the profile.
}
```

3. *Loading Appointments*: Appointment objects in Outlook are stored as an indexed list so loading them is a simple matter of opening the MAPI folder and iterating through the items in the calendar folder. This is accomplished as follows

```
Outlook::MAPIFolderPtr pMapiFolder;
using namespace Outlook;
pMapiFolder = pNamespace->GetDefaultFolder(olFolderCalendar);

//get the items in the folder
Outlook::_ItemsPtr pItems = pMapiFolder->GetItems();
int nApptCount = pItems->GetCount();
Outlook::_AppointmentItemPtr pApptItem;

for(int i=1;i<=nApptCount;i++)
{
   COleVariant covItemIndex((short)i);
   pApptItem = (_AppointmentItemPtr)pItems->Item(covItemIndex)
   COleDateTime odtStartTime(pApptItem->GetStart ());
   COleDatTime odtEndTime(pApptItem->GetEnd ());
   CString lpszSubject = (LPCSTR)pApptItem->GetSubject();
   CString lpszBody = (LPCSTR)pApptItem->GetBody();
   CString lpszLocation = (LPCSTR)pApptItem->GetLocation();
   long nDuration = pApptItem->GetDuration();
   long nReminder = pApptItem->GetReminderMinutesBeforeStart();

   //Get the Recurrence pattern

   if(pApptItem->IsRecurring)
   {
      Outlook::RecurrencePatternPtr pRecPat;
      pRecPat = pApptItem->GetRecurrencePattern();
      OlRecurrenceType nRecType;
      pRecPat->get_RecurrenceType(&nRecType);
      OlDaysOfWeek nDaysOfWeekMask;
      pRecPat->get_DayOfWeekMask(&nDaysOfWeekMask);
      long nInterval = pRecPat->GetInterval();
      long nInstance = pRecPat->GetInstance();
      COleDateTime odtPatStDate(pRecPat->GetPatternStartDate());
      COleDateTime odtPatStTime(pRecPat->GetPatternStartTime());
      COleDateTime odtPatEndDate(pRecPat->GetPatternEndDate());
      COleDateTime odtPatEndTime(pRecPat->GetPatternEndTime());
      long nDayOfMonth = pRecPat->GetDayOfMonth();
   }
}
```

Every appointment object contains some basic information such as the subject, body, start and end times, the duration, location and a list of attendees. If an appointment is a recurring event there will be additional information that specifies the recurrence pattern. The recurrence pattern can be used to determine when the next instance of the event will occur. Outlook supports six basic recurrence patterns:

1. Daily: These are events that occur every *nInterval* days or on every weekday (nDaysOfWeekMask set to all weekdays).
2. Weekly: These are events that occur every *nInterval* weeks on certain days of the week (*nDaysOfWeekMask* set to all days that this event occurs on).
3. Monthly: These are events that occur every *nDayOfTheMonth* day every *nInterval* months.

4. MonthlyN: These are events that occur every *nInstance* of a particular day (*nDaysOfWeekMask*) every *nInterval* months.
5. Yearly: These are events that occur once every *nDayOfMonth* of a particular month.
6. YearlyN: These are events that occur every *nInstance* of a particular day of a particular month.

Once every appointment is extracted from Outlook it is loaded into the L1 calendar store which provides facilities for searching the database and answering complex queries to determine whether a certain time slot will be free.

4. *Shutting Down*: As with another COM application it is necessary to release all objects instantiated and unintialize COM. In RCal the following steps logoff the MAPI profile and release all the objects that were instantiated.

```
pNamespace->Logoff();
pOutlook.Release();
```

It is not necessary to uninitialize COM when using MFC because the framework calls `OleUnInitialize()` when the application terminates. However if `CoInitialize(NULL)` was used to initialize COM then it will be necessary to uninitialize COM with a call to `CoUninitialize()`.

4.2 Level One Calendar Database

The Outlook calendar database is advantageous because it provides persistent storage of data between lifecycles of the agent as well as a well adopted user interface for updating the database. However accessing this store can be slow. In addition to this there are no facilities for querying the calendar as well.

To overcome these hurdles RCal uses an in-memory store for the calendar data that is initially populated with the contents of the Outlook calendar. This data structure – the appointment store - is a linked list of appointment objects. An appointment object is a structure that consists of the following fields.

1. Subject: This is the subject of the appointment.
2. Body: This is the body of the appointment
3. Location: This is the location of the event.
4. Required Attendees: This is a list of required attendees that will participate in this event. The members of this list must agree to the constraints of this event in order for this event to be scheduled
5. Optional Attendees: This is a list of optional attendees that will participate in this event. The members of this list may or may not agree to the constraints of the event and will not be considered while scheduling meetings
6. Start Time: This is the start time of this event and is a fully qualified value that consists of the date and time in 12 hour format along with the AM/PM identifier and any optional time zone identifier.

7. End Time: This is the end time of the event and is a fully qualified value that consists of the date and time in 12 hour format along with an AM/PM identifier and any optional time zone identifier.
8. Time Stamp: This is the time stamp that indicates when this event will expire. This applies only to events scheduled as tentative and is set to NULL for events marked as busy.
9. Attribute: This is an attribute field that is used to hold any state data associated with this event. It is typically set to the Contract ID or the Bid ID of the message that cause this event to be scheduled depending upon whether this is a tentative event or not.
10. Busy Status: This flag indicate whether this event is a tentative slot that will expire or an event that is scheduled and marked as busy.
11. Duration: This is the duration of the event in minutes and indicates the length of time that this event takes before completion.
12. Reminder: This is the delay in minutes (if any) before the start of the event before a reminder must be generated.
13. Reminder Flag: This flag indicates whether this event will generate a reminder prior to its start.
14. Outlook Index: This is the index into the permanent data store that this event is stored for purposes of synchronization with Outlook.
15. Recurrence Flag: This flag indicates whether this event is a recurring event and has any recurrence data associated with it.
16. Recurrence Data: This data structure holds information relate to the recurrence pattern of this event if applicable.

These fields maintain the basic information about an event and are all primitive data types such as strings or integers. The last field – Recurrence Data – is a complex structure that holds the recurrence pattern of this event if the recurrence flag is set indicating that this is a recurring event. The fields of this structure are:

1. Status: The flag indicates the valid status of this recurrence data structure and can be one of two values - __RD_VALID__ or __RD_NULL__
2. Recurrence Type: This is the recurrence type that indicates the type of recurrence pattern that this event follows. The values for this field can be:
    a. __RT_DAILY__: This is a daily recurrence
    b. __RT_WEEKLY__: This is a weekly recurrence
    c. __RT_MONTHLY__: This is a monthly recurrence.
    d. __RT_MONTHLYN__: This is an N Monthly recurrence
    e. __RT_YEARLY__: This is a yearly recurrence
    f. __RT_YEARLYN__: This is an N Yearly Recurrence
3. Recurrence Interval: This is the interval (if any) of this recurrence pattern. This value indicates how often this event repeats.
4. DayOfWeekMask: This is a bit mask that indicates the days (if any) to which this recurrence pattern applies.
5. Instance: This is the instance (if any) of the day of the month. This value indicates if this event occurs on a particular instance (first, second etc) of a day of a month.

6. Start Time: This is the time when the recurrence starts. If this is not specified then it is set to the start time of the first occurrence of the event.
7. Start Date: This is the date when the recurrence starts. If this value is not specified then it is set to the date of the first recurrence.
8. End Time: This is the time when the recurrence pattern ends. If the pattern does not end then this value is set to NULL.
9. End Date: This is the date when the recurrence pattern ends. If the pattern does not end then this value is set to NULL.
10. End Flag: This value indicates whether the recurrence pattern ends.

4.3 Searching Calendars

The two main requirements that must be fulfilled before RCal can accurately schedule meetings on behalf of its user are (a) RCal must have an up-to-date copy of the users current calendar and (b) it must be possible to search this calendar for free time slots and answer queries that verify whether a certain time slot is free.

The former condition is satisfied by simply importing the user's calendar from Outlook and ensuring that the Level One store is synchronized with Outlook. The second requirement is fulfilled by the searching capabilities of the level one calendar store. The two main queries that need to be executed are (a) checking whether a particular time is free and (b) finding a free time slot of a given duration. Both of these queries must take recurring appointments into account.

4.4 Querying a Time Instance

The level one calendar store holds all the scheduled events in an indexed list. Determining whether a particular time is free requires iterating through this list and checking whether the specified time lies with the instance of any of the scheduled events. In addition to checking for standard events the algorithm must also check whether the time lies within any instance of any recurring appointments. The algorithm for examining a recurrence pattern to check whether a specified date lies on the instance of a recurring appointment is as follows.

```
BOOL OccursOn(recurrencePattern, date)
      returns true if the specified date lies on this recurrence pattern

   if(recurrencePattern->startDate > date)
      return false;

   if(recurrencePattern->Ends())
      if(recurrencePattern->endDate < date)
            return false;
   nStartYear = GetYear(recurrencePattern->startDate)
   nStartMonth = GetMonth(recurrencePattern->startDate)
   nDate = GetDate(date)
   nMonth = GetMonth(date)
   nYear = GetYear(date)
   switch(recurrencePattern->recurrenceType)
      case __RT_DAILY__
         if(recurrencePattern->nDaysOfWeekMask == everyWeekday)
```

```
      if(GetDayOfWeek(date) is a weekday)
         return true else return false
   if(recurrencePattern->nDayOfWeekMask == NoDaySpecified)
      if((date – recurrencePatter->startDate) % nInterval == 0)
         return true else return false
   break
case __RT_WEEKLY__
   if(GetDayOfWeek(date) in recurrencePattern->nDayOfWeekMask)
      leadDays = GetDayOfWeek(recurrencePattern->startDate)-1
      d1 = recurrencePattern->startDate – leadDays
      leadDays = GetDayOfWeek(date)-1
      d1 = d1 + leadDays
      totalDays = date – d1
      if(totalDays % (7*nInterval)) == 0)
         return true else return false
   break
case __RT_MONTHLY__
   if(GetDay(date) != nDayOfMonth)
      return false
   if(nMonth+12*(nYear-nStartYear)–nStartMonth)%nInterval) == 0)
      return true else return false
   break
case __RT_MONTHLYN__
   if(nMonth+12*(nYear-nStartYear)–nStartMonth)%nInterval) != 0)
      return false
   if(GetDayOfWeek(date) not in recurrencePattern->nDayOfWeekMask)
      return false
   if(IsWeekDay(recurrencePattern->nDaysOfWeekMask))
      if(IsWeekDayInstance(date, recurrencePattern->nInstance)
         return true else return false
   if(IsWeekendDay(recurrencePattern->nDaysOfWeekMask))
      if(IsWeekendDayInstance(date, recurrencePattern->nInstance)
         return true else return false
   if(IsDay(recurrencePattern->nDaysOfWeekMask))
      if(IsDayInstance(date, recurrencePattern->nInstance)
         return true else return false
   break
case __RT_YEARLY__
   if(nMonth == nStartMonth)
      if(GetDay(date) == GetDay(recurrencePattern->startDate))
         return true else return false
   return false
   break
case __RT_YEARLY_N__
   if(nMonth != nStartMonth)
      return false
   if(IsWeekDay(recurrencePattern->nDaysOfWeekMask))
      if(IsWeekDayInstance(date, recurrencePattern->nInstance)
         return true else return false
   if(IsWeekendDay(recurrencePattern->nDaysOfWeekMask))
      if(IsWeekendDayInstance(date, recurrencePattern->nInstance)
         return true else return false
   if(IsDay(recurrencePattern->nDaysOfWeekMask))
      if(IsDayInstance(date, recurrencePattern->nInstance)
         return true else return false
   break
```

The complete algorithm for determining whether a particular time is free involves iterating through all the scheduled appointments and checking if the specified time lies

within any occurrence of any of them. This algorithm given below uses `OccursOn()` to determine whether the specified time lies within an occurrence of a recurring appointment.

```
BOOL IsFree(dateTime)
   foreach appointment in calendar
      startTime = GetStartTime(appointment)
      startDateTime = GetStartDateTime(appointment)
      endTime = GetEndTime(appointment)
      endDateTime = GetEndDateTime(appointment)
      time = GetTime(dateTime)
      if(appointment is recurring)
         if((time >= startTime) && (time < endTime))
            if(OccursOn(dateTime, GetRecurrencePattern(appointment)))
               return false
      if((dateTime >= startDateTime) && (dateTime < endDateTime))
         return false
   return true
```

4.5 Finding Free Time Slots

The appointment store data structure is a linked list of appointment objects that indicate the timespan of every event that is scheduled in the user's calendar. Finding a free time slot is an iterative process of generating an initial time slot based on the given constraints and checking if that time slot is free. If the slot is not free then the process is repeated with a new slot that is incrementally increased. The algorithm for this simple process is given below.

```
TimeSlot GetFreeTimeSlot(duration, lowerBound, upperBound)
   timeslot = CreateNewTimeSlot(duration, lowerBound)
   while(GetEndTime(timeslot) < upperbound)
      if(IsFree(timeslot))
         return timeslot
      Increment(lowerBound)
      timeslot = CreateNewTimeSlot(duration, lowerBound);
   return NULL;
```

The `IsFree(timeslot)` method used in the above algorithm is a variation of the `IsFree(dateTime)` method shown above that checks whether the specified timeslot is free rather than just the specific time. The `GetFreeTimeSlot` method finds a free time slot of the given duration with the additional constraints that it does not begin before the date and time specified by `lowerBound` and does not end after the time specified by `upperBound`. The `Increment(lowerBound)` method is called to increment the `lowerBound` so that the next time slot of the given duration can be generated.

5. The Distributed Meeting Scheduling Engine

The distributed meeting scheduling engine is responsible for carrying out the various steps involved in scheduling a meeting. This component of RCal integrates with the communication layer to accept inputs from its peers and make decisions based on this data. It integrates with the level one calendar data store to access local calendar information for the attendee it represents. This calendar information is used when responding to meeting requests made by other RCal agents and when updating the calendar store with new meetings that have been scheduled.

The actual steps involved and the order in which they are executed depends upon the status of the agent in the current negotiation. If the agent is a host for the meeting then it is responsible for initiating the process and the first step is to generate a contract that must be sent out to all the attendees for that particular meeting. In this case the agent is actively scheduling a meeting and is an active participant in the negotiation process.

If the agent is not a host in the negotiation then it only responds to contracts issued by a host. In this case it will process the contract issued, evaluate the constraints and parameters and check the local calendar data store before responding with a bid that accepts the contract, offers alternatives or rejects the contract outright.

It should be noted that agents that are hosts in a negotiation do not present bids or confirmations for that negotiation while agents that are not hosts do not generate contracts or awards. This difference in interaction status and scenarios is very important since they form two complementary sides of the distributed problem solving engine formed by the cooperating and coordinating agents.

5.1 Generating Contracts

A new meeting is initiated when a human user directs his/her calendar agent to schedule a meeting with a specified set of other users. This initial request (figure 4) solicits information from the user such as the meeting time, the constraints of the meeting, the location and the other information.

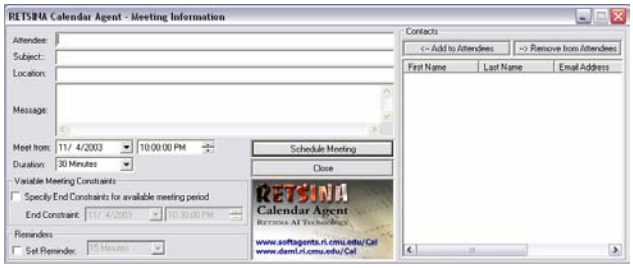


Figure 4: User Interface for Scheduling a new meeting

This information is used to generate a contract that is sent to the RCal agents representing other attendees. RCal uses the Knowledge Query Manipulation Language (KQML) [27] as its base language for data and knowledge representation. A contract consists of the following main fields

1. Contract ID: A Unique Identifier for this contract
2. Negotiation ID: A Unique identifier for this negotiation

3. Attributes: Details of the meeting to be scheduled. This is typically data that does not affect the outcome of the negotiation
4. Constraints: The constraints of the meeting such as the start time, duration, end time and the location of the meeting. This is information that is critical for making decisions regarding the outcome of the meeting.

If the constraints of the meeting do not specify an end time then it is assumed that the meeting must be scheduled to start at the specified start time and should last for the duration of time specified by the duration parameter. In other words the absence of an explicit end time implies a meeting end time given by the start time plus the duration. In this case the only valid bids are ones that accept or reject the solicited meeting time. The structure of such a contract is as follows.

```
Type: Contract
Contract ID: 9861821826
Negotiation ID: 9125251782
Expires: 9/4/2003 11:57:00 AM
Attributes
   Subject: "Event Subject"
   Body: "Event Body"
   Location: "Event Location"
   Attendees: "Attendees"
Constraints:
   Start: 9/15/2003 10:00:00 AM
   Duration: 30
```

If the end time is specified then the meeting contract is more flexible and calls for bids that specify meeting times that start sometime after (or at) the start time, end before (or at) the specified end tine and last for the specified duration. Such contracts (example given below) are the results of requests such as “Schedule a half hour meeting sometime in the next week”.

```
Type: Contract
Contract ID: 9861821826
Negotiation ID: 9125251782
Expires: 9/4/2003 11:57:00 AM
Attributes
   Subject: "Event Subject"
   Body: "Event Body"
   Location: "Event Location"
   Attendees: "Attendees"
Constraints:
   Start: 9/15/2003 10:00:00 AM
   End: 9/30/2003 5:00:00 PM
   Duration: 30
```

5.2 Presenting Bids

Once a meeting request has been broadcast by a host in the form of a contract the attendees receiving the contract must respond with bids that accept the meeting, reject it outright or reject it and offer alternatives. The types of bids that are applicable depend upon the type of contract that is generated.

A bid must contain the following main fields of information

1. Contract ID: The ID of the contract that this bid is in reply to
2. Negotiation ID: The unique identifier of the negotiation that this bid corresponds to
3. Bid ID: The unique ID of the bid
4. Attributes: The attributes of the meeting as specified in the contract
5. Status: The status of the bid – either accept, reject or alternate depending upon the type of contract and the decision of the bidder
6. Expires: The absolute time at which the bid expires

For fixed constraint contracts the bidders have the option of either accepting the meeting time specified or rejecting it outright. Offering alternatives is not an option since the meeting contract does not solicit them. The structure of such a bid is shown below.

```
Type: Bid
Contract ID: 9861821826
Negotiation ID: 9125251782
Bid ID: 25821391289
Expires: 9/4/2003 11:59:00 AM
Status: Accept
Attributes
   Subject: "Event Subject"
   Body: "Event Body"
   Location: "Event Location"
   Attendees: "Attendees"
Constraints:
   Start: 9/15/2003 10:00:00 AM
   Duration: 30
```

In cases where the contract is flexible, the attendees may either reject the contract – indicating that they do not wish to participate – or they may offer available time slots at which they are available to meet. In this case the option of merely accepting the contract is not applicable since no explicit meeting time is specified in the contract. A bid that that rejects the contract and offers alternatives is structured as follows.

```
Type: Bid
Contract ID: 9861821826
Negotiation ID: 9125251782
Bid ID: 25821391289
Expires: 9/4/2003 11:59:00 AM
Status: Alt
Alternates:
   T1: 9/15/2003 11:00:00 AM
   T2: 9/17/2003 10:00:00 AM
   T3: 9/17/2003 11:00:00 AM
   T4: 9/17/2003 1:00:00 PM
   T5: 9/17/2003 4:00:00 PM
Attributes
   Subject: "Event Subject"
   Body: "Event Body"
   Location: "Event Location"
   Attendees: "Attendees"
Constraints:
   Start: 9/15/2003 10:00:00 AM
   End: 9/30/2003 5:00:00 PM
   Duration: 30
```

Bids that accept contracts with a window of time are currently disabled in the RCal system but it may be applicable policy to allow such bids that implicitly state that the attendee has no meeting preferences and is available at all and any times. In such a case system policy may be set to indicate that such attendees do not constrain the meeting scheduling process and should not be considered when evaluating applicable time slots. Once a time is selected, they merely need to be notified and will accept.

Once a bid is made its is the responsibility of the bidder to mark these times as tentative in the local calendar store in order to avoid offering the same time slots to other hosts. A bid on a contract is a binding agreement that must be upheld in order for the meeting scheduling process to succeed. However it may be the case that certain times are highly sought after in multiple meeting negotiations with multiple hosts. To avoid the possibility of tentative meeting slots taking up the calendar, each bid has an expiry time associated with it and it is the responsibility of the bidder to monitor this time and make the time slot available to other hosts once the expire time has passed. It is the responsibility of the host of the meeting to check this expiry time and process all the bids before they expire. In cases where the bids expire before they are evaluated by the host, the meeting scheduling process fails and must be abandoned.

5.3 Evaluating Bids

After a host has broadcast contracts to all the attendees specified by the user, and received the bids it must evaluate all bids for applicability and to determine whether a suitable meeting time exists. Searching for the earliest meeting time that has been suggested by all the attendees is a simple matter of sorting all the times suggested by each attendee in ascending order and then iterating through the times suggested by an attendee and checking if has been offered by all the attendees. The algorithm for this is given below.

```
Time EvaluateBids(Bids)
   SortAscending(Bids)
   foreach(time t1 offered by Attendee[0])
      found = false
      for(i=1;i<totalAttendees;i++)
         if(t1 offered by Attendee[i])
            found = true
         else
            break
      if(found = true)
         return time
      else
         GenerateNewContract()
```

The algorithm above finds the earliest time that has been suggested by all the attendees. In case a time is not found then a new contract is generated and broadcast and more times are solicited. New contracts can only be generated for negotiations that begin with initial contracts that specify an end time, i.e. contracts that provide a flexible time window within which the meeting must be scheduled. In such cases new contracts should be generated in manner that takes into account the latest time suggested by an attendee. Generating a new contract assumes that no mutually acceptable meeting time was found

and so the start time of the new contract must be the end time of the latest time suggested amongst all the times suggested by all the attendees. The algorithm for this is give below.

```
Contract GenerateNewContract()
   startTimeConstraint = NULL
   foreach(attendee in Attendees)
      latestSlot = GetLatestSlot(attendee)
      if(latestSlot > startTimeConstraint)
         startTimeConstraint = latestSlot
   endTimeConstraint = GetEndTimeConstraint(oldContract)
   if(startTimeConstraint >= endTimeConstraint)
      return false
   else
      return NewContract(startTimeConstraint, endTimeConstraint)
```

The above algorithm generates a new contract with a new start time constraint. This new contract will have a new contract ID but will carry the negotiation ID from the old contract since they are part of the same negotiation.

5.4 Awards

Once a suitable meeting time is found that is acceptable to all the attendees, an award indicating this time is generated and broad cast to all the attendees. The host also marks the meeting as tentatively scheduled in the local calendar store. An award does not mark the end of the negotiation and is not a guarantee that the meeting will be scheduled. The attendees reserve the option to reject an award and retract a bid previously offered. In this case the negotiation may be assumed to have failed and be abandoned or it may be restarted with a new contract. The outcome in such a situation depends upon system policy. An award sent out to all the attendees consists of the following fields:

1. Contract ID: Unique identifier of the contract that is awarded
2. Bid ID: Unique ID of the Bid that was submitted
3. Award ID: A unique identifier for this award
4. Attributes: The attributes of the meeting such as the subject, body and location
5. Constraints: The constraints of the meeting such as the start time and the duration

An award contains the details of the selected meeting time in the constraints field. The structure of an award is as follows.

```
Type: Award
Contract ID: 9861821826
Negotiation ID: 9125251782
Bid ID: 25821391289
Expires: 9/4/2003 11:59:00 AM
Status: Accept
Attributes
   Subject: "Event Subject"
   Body: "Event Body"
   Location: "Event Location"
   Attendees: "Attendees"
Constraints:
   Start: 9/15/2003 10:00:00 AM
   Duration: 30
```

The negotiation ID indicates the negotiation that this award is a part of while the bid ID and contract ID specify the contract and the bid that are being awarded.

5.5 Tentative Scheduling

It is possible for multiple meetings to be concurrently scheduled with various attendees, some of which may be participating in more than one negotiation. It is imperative to ensure that time slots allocated as available to a particular host not be submitted for consideration to another host as well. This will ensure that there are no conflicts in meetings scheduled.

Each agent that submits a bid to the host of that negotiation keeps a record of the times suggested along with an identifier that uniquely identifies that contract being bid upon. In addition to this, each time slot suggested is marked as tentatively scheduled and is associated with a time stamp that indicates the absolute time at which the offer of availability of that time slot will expire. If the bid is not evaluated within that time period that time slot is marked as free and will be offered to other hosts requesting it.

Once a particular host evaluates all bids and submits awards, the bidder in turn checks its local registry for the time slot represented in the award. The bidding agent then accepts the award only if all the following conditions are met:

1. The Time slot is marked as free or tentative
2. The Contract ID on the award is the same as the contract ID specified with the time slot
3. The Time slot offer has not expired.

RCal iterates through its local store of recent bids and examines all the expiry times associated with all the time slots suggested and removes all tentatively scheduled spots and marks them as free.

Once a suitable meeting time is found and all the conditions for validity of the award are satisfied then the bidder marks that time in it calendar as busy and deletes all tentative time slots that with the contract ID for that negotiation. In the event that the time slot has expired, the bidder removes all the tentative slots associated with that contract and rejects the award by returning a failed confirmation message.

5.6 Confirmations

Confirmations are sent by bidders in response to awards that are received from meeting hosts. A confirmation message is the last exchange in a negotiation and signifies that the negotiation is complete. In the event that a bidder accepts the award it receives it sends out a confirmation with a status flag set to success to indicate that the timeslot awarded has been marked as busy and the meeting has been scheduled.

A confirmation message must contain the following fields as given below.

1. Contract ID: Unique ID of the contract being confirmed
2. Bid ID: Unique ID of the bid being confirmed.
3. Award ID: Unique ID of the award being confirmed
4. Confirm ID: Unique ID of this confirm message.
5. Constraints: Constraints of the meeting such as the start time and the duration
6. Status: Status flag of this confirmation that accepts or rejects the award

```
Type: Confirm
Contract ID: 9861821826
Negotiation ID: 9125251782
Bid ID: 25821391289
Expires: 9/4/2003 11:59:00 AM
Status: Accept
Attributes
   Subject: "Event Subject"
   Body: "Event Body"
   Location: "Event Location"
   Attendees: "Attendees"
Constraints:
   Start: 9/15/2003 10:00:00 AM
   Duration: 30
```

If the status flag has been set to reject then the bidder is indicating that the award has been rejected and the meeting cannot be scheduled. This may be due to any number of factors such as the bid having expired. In such an event it is the responsibility of the host to evaluate this response and to notify the other attendees of the status of the meeting. The meeting may be cancelled and the negotiation abandoned or the meeting may still be marked as scheduled with certain attendees declining participation. The actual response of the system will depend upon the system policy in effect.

6. Communication and Connectivity

RCal is a P2P communication architecture based on the RETSINA agent framework. Every RCal agent registers its name with an agent name server (ANS) that maps the agent name to a *host:port* network location record. Agents that wish to contact this agent can then lookup the agent name on the ANS and then open a communication socket to that host and port. The communication is carried out by the RETSINA communicator [26] – a component provided by the RETSINA [25] agent framework. The Communicator manages connections and communication state and is responsible for passing data back and forth between the application layer and the agent communication layer.

6.1 Message Specifications

RCal processes requests and returns results packaged into messages that encode control information and data relevant to the communication. RCal uses the Knowledge Query Manipulation Language (KQML) [27] for agent communication. The details of the messages sent and received are as follows:

1. Contract: This message is sent when announcing contracts to attendees of a new meeting. The content of the message contains the details of the contract such as the identifiers, attributes and constraints related to the contract.

```
(tell
   :sender (kingtiny_cs.cmu.edu-CalendarAgent)
   :receiver (kingtiny2_cs.cmu.edu-CalendarAgent)
   :ontology (default-ontology)
   :language (default-language)
   :reply-with (192837129812893312630)
   :in-reply-to ()
   :forward-to ()
   :content (tell
         :type (contract)
         :contractID (9861821826)
         :negID (9125251782)
         :expires (9/4/2003 11:57:00 AM)
         :attributes (tell
                           :subject (Event Subject)
                           :body (Event Body)
                           :location (Event Location)
                           :attendees (Attendees))
         :constraints (tell
                           :start (9/15/2003 10:00:00 AM)
                           :duration (30))))
```

2. Bid: This message is sent by an attendee as a response to a contract. Depending upon the type of contract this message specifies the status of the attendee with respect to the received contract. This message may specify that the attendee accepts the contract, rejects the offered time or is offering alternative times to meet. The content

of the message contains the details of the bid such as the identifiers, the status of the bid and any alternate times offered.

```
(tell
   :sender (kingtiny2_cs.cmu.edu-CalendarAgent)
   :receiver (kingtiny_cs.cmu.edu-CalendarAgent)
   :ontology (default-ontology)
   :language (default-language)
   :reply-with (19283712981289331263O)
   :in-reply-to ()
   :forward-to ()
   :content (tell
               :type (bid)
               :negID (9125251782)
               :contractID (9861821826)
               :bidID (981251925265)
               :status (accept)
               :duration (30)
               :attributes (tell
                           :subject (Event Subject)
                           :body (Event Body)
                           :location (Event Location)
                           :attendees (Attendees))
               :constraints (tell
                           :start (9/15/2003 10:00:00 AM)
                           :duration (30))))
```

3. Award: This is a message that is sent by the host once all bids have been evaluated and a meeting time has been found.

```
(tell
   :sender (kingtiny_cs.cmu.edu-CalendarAgent)
   :receiver (kingtiny2_cs.cmu.edu-CalendarAgent)
   :ontology (default-ontology)
   :language (default-language)
   :reply-with (19283712981289331263O)
   :in-reply-to ()
   :forward-to ()
   :content (tell
               :type (award)
               :negID (9125251782)
               :contractID (9861821826)
               :bidID (981251925265)
               :attributes (tell
                           :subject (Event Subject)
                           :body (Event Body)
                           :location (Event Location)
                           :reqAttendees (Required Attendees)
                           :optAttendees (Optional Attendees))
               :constraints (tell
                           :start (9/15/2003 10:00:00 AM)
                           :duration (30))))
```

4. Confirm: The confirmation message is sent by an attendee to notify the host of the confirmation of the award and to indicate that the meeting has been scheduled. The confirm message is the final message that is exchanged in the negotiation and contains a status flag that indicates whether the attendee was able to mark the user's calendar with the details of the new meeting.

```
(tell
   :sender (kingtiny2_cs.cmu.edu-CalendarAgent)
   :receiver (kingtiny_cs.cmu.edu-CalendarAgent)
   :ontology (default-ontology)
   :language (default-language)
   :reply-with (192837129812893312630)
   :in-reply-to ()
   :forward-to ()
   :content (tell
               :type (award)
               :negID (9125251782)
               :contractID (9861821826)
               :bidID (981251925265)
               :status (accept)
               :attributes (tell
                           :subject (Event Subject)
                           :body (Event Body)
                           :location (Event Location)
                           :reqAttendees (Required Attendees)
                           :optAttendees (Optional Attendees))
               :constraints (tell
                           :start (9/15/2003 10:00:00 AM)
                           :duration (30))))
```

5. Schedule Request: This message is sent by the E-Secretary agent to query RCal for user availability at a particular time.

```
(tell
   :sender (esec-811271928329)
   :receiver (kingtiny_cs.cmu.edu-CalendarAgent)
   :ontology (default-ontology)
   :language (default-language)
   :reply-with (1954645693312630)
   :in-reply-to ()
   :forward-to ()
   :content (tell
               :command (request_appt)
               :fname (Rahul)
               :lname (Singh)
               :email (kingtiny@cs.cmu.edu)
               :subject (Test Appointment)
               :body (Test Appointment)
               :location (NSH 1617)
               :startTime (10/25/2003 10:30:00 AM)
               :duration (30)
               :targetRealName (Rahul)
               :targetEmailAdd (kingtiny@cs.cmu.edu)))
```

RCal responds with a simple message that indicates whether the user is free at that time or not.

```
(tell
   :sender (kingtiny_cs.cmu.edu-CalendarAgent)
   :receiver (esec-811271928329)
   :ontology (default-ontology)
   :language (default-language)
   :reply-with (1954645693312630)
   :in-reply-to ()
   :forward-to ()
   :content (tell
               :startTime (10/25/2003 10:30:00 AM)
               :duration (30)
               :status  (accept))
```

If the user is not free and system policy is set to respond with alternate times then RCal responds with the following message.

```
(tell
   :sender (kingtiny_cs.cmu.edu-CalendarAgent)
   :receiver (esec-811271928329)
   :ontology (default-ontology)
   :language (default-language)
   :reply-with (1954645693312630)
   :in-reply-to ()
   :forward-to ()
   :content (tell
               :startTime (10/25/2003 1:30:00 PM)
               :duration (30)
               :status  (alt))
```

6. Schedule Confirm: This message is sent by the E-Secretary agent to schedule a meeting at the time specified. This time is determined to be free by a previous schedule request message.

```
(tell
   :sender (esec-811271928329)
   :receiver (kingtiny_cs.cmu.edu-CalendarAgent)
   :ontology (default-ontology)
   :language (default-language)
   :reply-with (1954645693312630)
   :in-reply-to ()
   :forward-to ()
   :content (tell
               :command conf_appt
               :start (tell
                           :day (25)
                           :month (10)
                           :year (2003)
                           :hour (10)
                           :min (30))
               :email (kingtiny@cs.cmu.edu)
               :subject (Test Appointment)
               :body (Test Appointment)
               :location (NSH 1617)
               :duration (30)
               :requiredAttendees (Rahul Singh)
               :optionalAttendees ()
               :targetRealName (Rahul Singh)
               :targetEmailAdd (kingtiny@cs.cmu.edu)))
```

RCal responds to this message with a simple message that indicates success or failure

```
(tell
   :sender (kingtiny_cs.cmu.edu-CalendarAgent)
   :receiver (esec-811271928329)
   :ontology (default-ontology)
   :language (default-language)
   :reply-with (1954645693312630)
   :in-reply-to ()
   :forward-to ()
   :content (tell
               :status (success)))
```

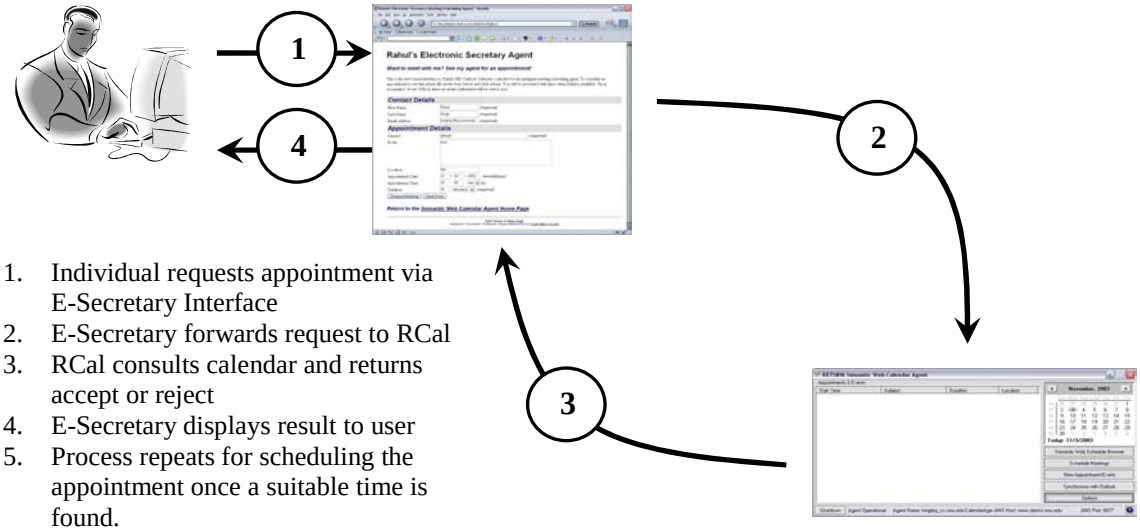
7. Extensions

The agent framework that RCal is based on allows for numerous extensions to be added to the system with minimal work. The two main extensions that are currently available are the web based E-Secretary agent and the appointment notification feature.

7.1 E-Secretary

Figure 5: The E-Secretary Agent

There may be occasions where users do not have access to their RCal agent, or may not use a PIM to manage their calendar and hence need some alternative mechanism for requesting meetings with people they know. Alternatively, individuals or organizations (such as clinics or dental surgeries) may want to publish a web-based interface where appointments can be requested. The RETSINA E-Secretary agent (FIGURE 5) is a web-based agent that facilitates appointment-requests, without the need for both parties to use RCal. The design is based on the concept of scheduling meetings via a human secretary – whereby the secretary interacts with meeting requesters, and manages meeting requests according to when time slots are available. In such cases, the negotiation is limited to only two parties – a human requesting a meeting or appointment at a preferred time, and RCal responding to this request (via the E-Secretary) with the appropriate meeting time based on the requestee's calendar. Typically, a user who manages their calendar using RCal also has an E-Secretary agent running on their behalf on the web.



1. Individual requests appointment via E-Secretary Interface
2. E-Secretary forwards request to RCal
3. RCal consults calendar and returns accept or reject
4. E-Secretary displays result to user
5. Process repeats for scheduling the appointment once a suitable time is found.

Figure 6: Interaction between E-Secretary and RCal

Anyone desiring a meeting enters details such as their name, email, location for the meeting and desired time via a form presented by the E-Secretary. The E-Secretary then sends the meeting request to RCal, which looks for an appropriate time slot. If one is found it is presented to the human requesting the meeting who can either accept or reject the proposed meeting time. If accepted, the E-Secretary sends a confirmation to RCal, which updates the calendar, and notifies both parties of the scheduled meeting via email.

The E-Secretary agent sends and receives messages in two categories.

1. Query: This is a message that is sent to RCal to determine whether the user is free at the time requested by an individual. The response to this message is an accept status code if the user is free and an alt status code along with an alternative time if the user is not free. If meeting scheduling via the E-Secretary interface is disabled or if access control lists are in effect then RCal will reject all requests with a reject status code. Figure 7 below shows the content of the message sent by the E-Secretary and Figure BLAH shows the message received in response.

```
(tell
  :command (request_appt)
  :fname (<firstName>)
  :lname (<lastName>)
  :email (<emailAddress>)
  :subject (<subject>)
  :body (<body>)
  :location (<location>)
  :startTime (<mm/dd/yyyy
              hr:min:00 AM/PM>)
  :duration (<duration in minutes>)
  :targetRealName (<targetRealName>)
  :targetEmailAdd (<targetEmailAdd>))
```

```
(tell
  :startTime (<mm/dd/yyyy
              hr:min:00 AM/PM>)
  :duration (<duration in minutes>)
  :status (<accept/reject/alt>))
```

Figure 7: Message sent (a) from E-Secretary to RCal and response received (b) when querying for a free time

The fields enclosed in angle brackets (<>) indicate data that is passed within that field. The `lastName`, `firstName` and `emailAddress` fields contain the last name, first name and email address of the requestee. The `targetRealName` field contains the first name to use for display purposes and the `targetEmailAddress` field contains the email address to send confirmation messages to. The `duration` field contains the duration of the appointment in minutes while the `startTime` is a fully qualified date and time. The `subject` and `body` fields give more information about the meeting.

The response message contains the status of the request that can be accept, alt or reject to indicate that the user is free, busy or does not wish to meet at the time specified in the `startTime` field. The `startTime` field contains a fully qualified date and time that the user is free at and the `duration` contains the length of time that the user can meet for.

2. Schedule: This is a message that is sent to RCal to task it to schedule an appointment in the user's calendar for the specified time. The response to this message is a success response that indicates that the meeting has been scheduled. Figure BLAH shows the content of the message that is sent and figure 8 shows the content of the message that is received by the E-Secretary in response.

```
(tell                                   (tell
  :command conf_appt                      :status (<success/failure>))
  :start (tell
         :day (<dd>)
         :month (<mm>)
         :year (<yyyy>)
         :hour (<hr>)
         :min (<min>))
  :email (<email>)
  :subject (<subject>)
  :body (<body>)
  :location (<location>)
  :duration (<duration in minutes>)
  :requiredAttendees (<attendees>)
  :optionalAttendees (<attendees>)
  :targetRealName (<targetRealName>)
  :targetEmailAdd (<targetEmailAdd>))
```

Figure 8: Message sent (a) from E-Secretary to RCal and response received (b) when scheduling an appointment.

The message sent once again contains the email address of the requestee and the subject, body, location, duration and start time of the appointment in the appropriate fields. The `requiredAttendee` and `optionalAttendee` fields are reserved for future use and must contain the full name of the requestee. The `targetRealName` field contains the first name of the requestee for display purposes and the `targetEmailAddress` field contains the email address to send confirmation messages to.

In cases where the security policy does not permit meetings to be scheduled via the E-Secretary interface or when access control lists are in effect then RCal will return a reject status code for all meeting requests made via the E-Secretary.

7.2 Appointment Notifications

Outlook generates reminders when an appointment is due. This reminder results in a dialog box that is displayed on the users desktop notifying the user of the impending event. RCal can trap these reminder events and forward them to a specified agent or dispatch them to a specified email address. In addition to this RCal can also generate its own notifications that notify the user of new meetings that have been scheduled.

The Outlook object model exposes outlook events via the `IConnectionPointContainer` interface of the ApplicationEvents object. The event sink can be setup by advising on the appropriate connection point as shown below.

```
HRESULT hr;

// Get server's IConnectionPointContainer interface.
IConnectionPointContainer* pCPC;
hr = pItem->QueryInterface(IID_IConnectionPointContainer,
                           (void **)&pCPC);
if (SUCCEEDED(hr))
{
      // Find connection point for events we're interested in.
      hr = pCPC->FindConnectionPoint(
                  __uuidof(Outlook::ApplicationEvents),
                  &m_pConnection);
      if (SUCCEEDED(hr))
      {
         hr = m_pConnection->Advise(static_cast<IDispatch*>(this),
                                    &m_dwCookie);
      }
      // Release the IConnectionPointContainer
      pCPC->Release();
}
```

Once the event sink is setup, outlook can notify RCal of the events as they occur via the Invoke method which can handle the events as shown below.

```
STDMETHODIMP CReminderHandler::Invoke(DISPID dispIdMember, REFIID riid,
LCID lcid, WORD wFlags, DISPPARAMS* pDispParams, VARIANT* pVarResult,
EXCEPINFO* pExcepInfo, UINT* puArgErr)
{
    if (IID_NULL != riid)
        return DISP_E_UNKNOWNINTERFACE;

      switch( dispIdMember )
      {
            case 0xf002:
                  AfxMessageBox("Caught Item Send Event");
                  break;
            case 0xf003:
                  AfxMessageBox("Caught New Mail event");
                  break;
            case 0xf004:
                  AfxMessageBox("Caught Reminder event");
                  break;
            case 0xf005:
                  AfxMessageBox("Caught Options Pages Add Event");
                  break;
            case 0xf006:
                  AfxMessageBox("Caught Startup event");
                  break;
            case 0xf007:
                  AfxMessageBox("Caught Quit event");
                  break;
      }
    return S_OK;
}
```

The dispatch ID indicates the type of event that has been fired and the switch statement executes code to handle that particular event.

RCal traps the reminder event from outlook and dispatches notifications to the user via a task agent that can handle the message delivery or via email. RCal can be configured to send different types of notifications via the interface shown in figure 9.

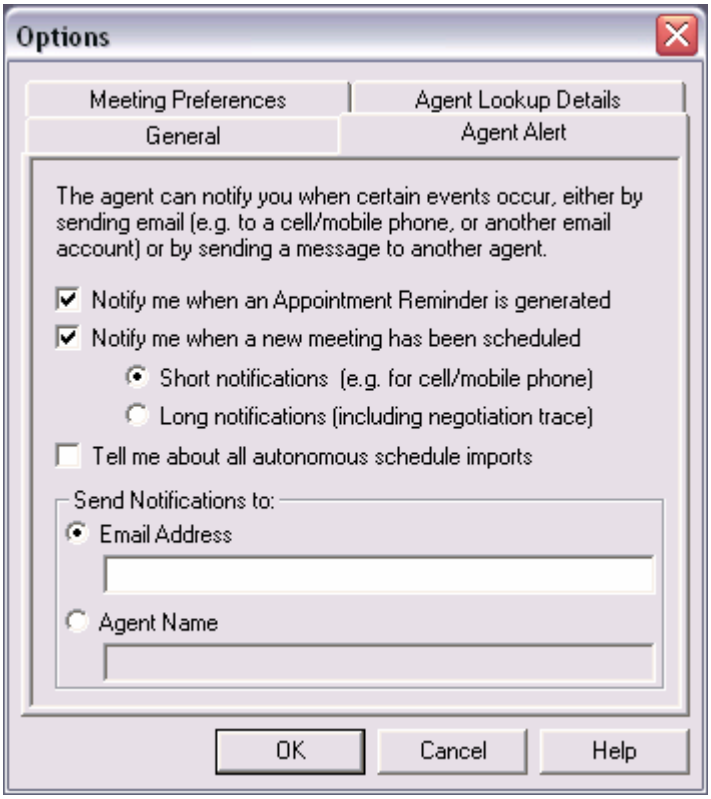


Figure 9: Event Notification configuration dialog

The two main notifications generated are:

1. Appointment Reminder: When an appointment reminder is generated RCal extracts the event details and formats it for dispatch. If RCal is configured to send the notification is send via email then the following message is sent to the email address specified.

   ```
   <subject>@<startTime> > <location>
   ```

   where `subject` is the subject of the event, `startTime` is the time at which the event starts and `location` is the location of the appointment. If RCal is configured to send the notification to an agent then the following KQML message is dispatched to the agent specified.

   ```
   (tell :command appointment_due
         :start (tell
                   :day (<dd>)
                   :month (<mm>)
                   :year (<yyyy>)
                   :hour (<hr>)
                   :minute (<min>)
   ```

```
                :second (<sec>))
        :subject (<subject>)
        :location (<location>))
```

2. Schedule Notification: RCal can also generate reminders when a new meeting has been scheduled. The notification can be short (for mobile devices) or long. The body of the short notification sent via email is

   ```
   <subject>@<startTime> > <location> [<firstname> <lastname>]
   ```

   and the body of the long notification is

   ```
   Firstname: <firstname>
   Lastname: <lastname>
   Meeting Subject: <subject>
   Meeting Body: <body>
   Meeting Starttime: <startTime>
   Meeting Duration: <duration>
   Meeting Location: <location>

   Meeting Attendees: <requiredAttendees>
                      <optionalAttendees>
   ```

8. Interoperability with the Semantic Web

One advantage that RCal enjoys over many other agent-based meeting scheduling systems is the ability to garner relevant information from the Semantic Web. Traditionally, calendar managers relied on humans to enter meetings that were not automatically entered. Whilst this approach works for occasional events, it breaks down when large-scale schedules (such as conference schedules) need to be added. In many cases, users simply enter single events to represent the whole schedule, and mark the time as busy, an approach that prohibits further negotiation during this time. For example, "*Bob*" might plan to attend the three-day "*Web Services Edge 2002 Conference*", and hence would want to update his calendar to reflect this. If this conference is entered as a single event, RCal will not schedule any meetings during this time. However, this does not accurately reflect Bob's actual schedule, as it does not take into account coffee and lunch breaks, and talks or presentations he may choose not to attend. In addition, Bob will not benefit from being able to consult his calendar to find out when individual events occur, or get reminders sent to his PDA or mobile phone. More importantly, it is often highly desirable to schedule meetings with other delegates at a conference, yet at such events, access to PIMs, schedules and the delegates themselves can be difficult.

RCal overcomes this problem by importing schedules directly from the Semantic Web [10]. Traditionally, extracting schedules from the World Wide Web has been a problem since HTML (currently used to publish schedules) requires custom-built software tools such as screen-scrapers to elicit the relevant information from the Web pages. Although XML representations can be used to simplify this, such an approach requires the standardized use of a single DTD or XML Schema. The Semantic Web relaxes this constraint by providing a framework (built upon XML) within which ontologies (formal specifications on how to represent concepts) can be built to describe concepts in the real world. Additionally, AI based reasoning techniques can be used within Web Services to search, translate, merge or navigate across markup in these ontologies.

The Semantic Web initiative utilizes RDF [11] to markup knowledge and publish it to the World Wide Web where software agents can access it. Moreover, information can be made available in a structured form with links to other pieces of information much in the way web pages are currently linked together. The Hybrid iCal[2] ontology, derived from the iCalendar specification [10], is one of several ontologies that provide a framework for schedules to be marked up in RDF and published on the web. Used by several applications (including RCal), it allows sharing and reasoning of schedules. Figure 10 illustrates a schedule marked up in RDF (the namespace declaration and <rdf:RDF> tags have been removed for brevity) with four events for the Web Services Edge 2002 West conference.

In this example, the iCal ontology is used to represent details about the events in the schedule, whereas the Dublin Core ontology[3] (xmlns:dc) is used to markup meta-

[2] For the Hybrid ICal Ontology see http://ilrt.org/discovery/2001/06/schemas/ical-full/hybrid.rdf
[3] For the Dublin Core Ontology see http://dublincore.org

information about the document itself such as the source, title and description. Fields in RDF can either be populated with simple text strings or with references to URIs (Uniform Resource Identifiers) that point to other RDF concepts such as the <dc:author> field in Figure 4. A URI reference such as this allows for more information to be extracted about the concept in question, which could be located elsewhere on the web. This framework leads directly from the present architecture of the World Wide Web except that now it links knowledge rather than plain text.

```
<!—WSESchedule.rdf -->
<ical:VCALENDAR rdf:ID="WebServicesEdge2002">
  <dc:source rdf:resource="http://www.sys-
    con.com/WebServicesEdge2002West/sched.cfm" />
  <dc:title>Web Services Edge 2002 West</dc:title>
  <dc:description>
    International Web Services Conference and Expo
  </dc:description>
  <dc:author rdf:resource="http://www.cs.cmu.edu/~kingtiny/kingtiny.rdf" />
  <ical:date>20020825</ical:date>
  <ical:VEVENT-PROP>
    <ical:VEVENT rdf:ID="Registration">
      <ical:DTSTART>
        <ical:DATE-TIME>
          <ical:TZID rdf:resource="#PDT" />
          <rdf:value>20021001T083000</rdf:value>
        </ical:DATE-TIME>
      </ical:DTSTART>
      <ical:DURATION>P0D0WT7H30M0S</ical:DURATION>
      <ical:LOCATION>
        San Jose McEnery Convention Center
      </ical:LOCATION>
      <ical:DESCRIPTION>Registration</ical:DESCRIPTION>
    </ical:VEVENT>
  </ical:VEVENT-PROP>
  <ical:VEVENT-PROP
    rdf:resource="http://www.daml.ri.cmu.edu/Schedules/WSE2002-
    JavaTrack.rdf#J1" />
  <ical:VEVENT-PROP
    rdf:resource="http://www.daml.ri.cmu.edu/Schedules/WSE2002-
    JavaTrack.rdf#J2" />
  <ical:VEVENT-PROP
    rdf:resource="http://www.daml.ri.cmu.edu/Schedules/WSE2002-
    JavaTrack.rdf#J3" />
</ical:VCALENDAR>
```

Figure 10: A schedule marked up in RDF

A schedule may contain multiple calendars (or schedules) represented by instances of the RDF class <ical:VCALENDAR>. Each calendar contains properties (<ical:VEVENT-PROP>) which link multiple events (<ical:VEVENT>) in the calendar. The schedule in Figure 4 contains four events of which the “Registration” event is inline while the others reference a URI to resources in another document (e.g. the http://www.daml.ri.cmu.edu/Schedules/WSE2002-JavaTrack.rdf#J1). Each event in the file WSE2002-JavaTrack.rdf contains information about the event such as its start time, duration etc exactly in the way the “Registration” event is marked up. Here the events

related to each track of the conference are in another document and the URIs allow an agent to navigate across this web of knowledge and extract information about them.

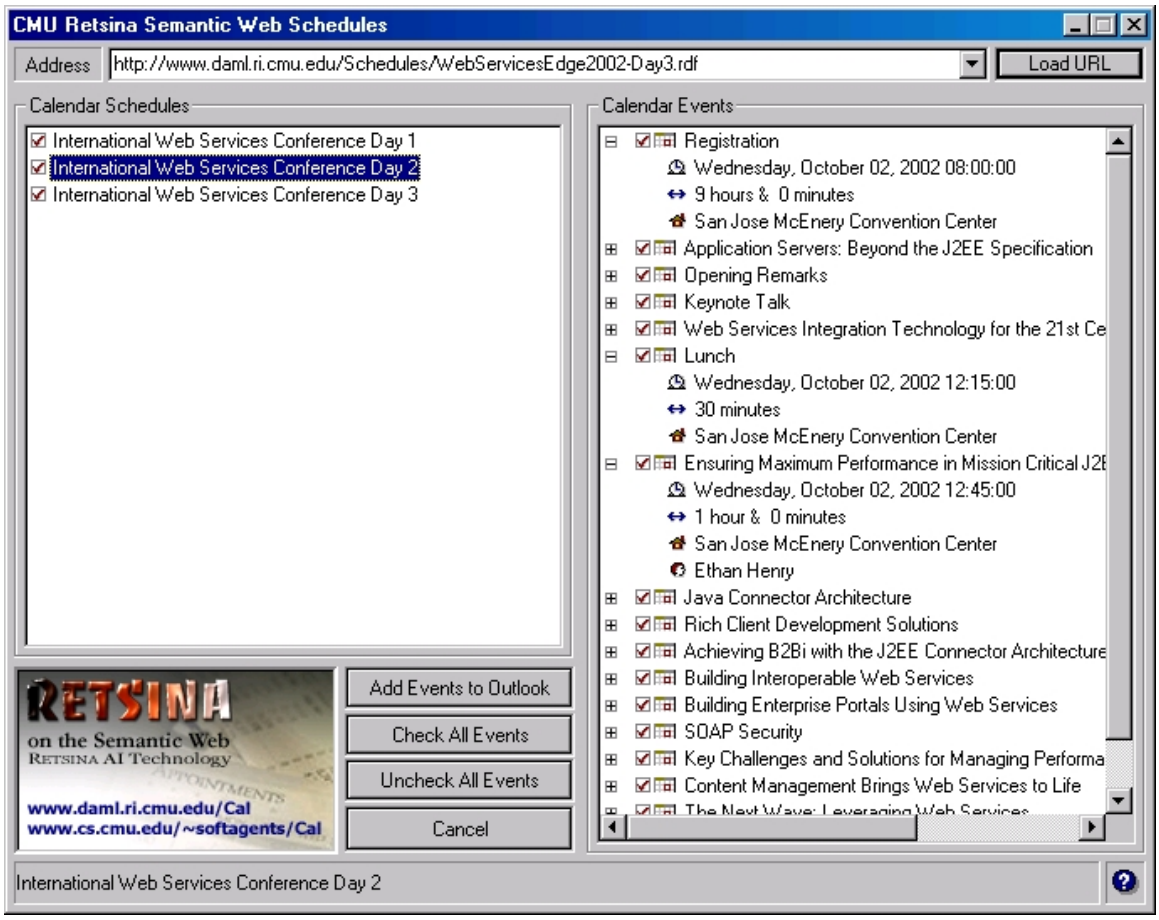


Figure 11: The Semantic Web Schedule Browser

RCal can import a schedule such as this using the Semantic Web Schedule Browser (Figure 11) and present the information to the user in an organized manner allowing more information to be retrieved by right-clicking on the concepts in question. Events can also be selected and imported into Outlook thus allowing a user to update his/her calendar without having to type out the details of each event.

9. Summary and Conclusions

RCal is a distributed problem solver that assists users in office environments in the domain of intelligent meeting scheduling. Built on a P2P architecture, RCal agents negotiate with each other on behalf of their users and decide on a common meeting time that is acceptable to all the participants. This protocol driven approach to intelligent meeting scheduling allows for a scalable and flexible system that can schedule multiple meetings concurrently and carry out negotiations with multiple agent at the same time.

RCal integrates with Outlook and uses Outlook's calendar as a persistent data store for the user's calendar. This integration also allows RCal to stay up-to-date on the user's schedule and notify the user of new appointments that are scheduled. Using Outlook's notification feature, RCal is able to dispatch notifications of new appointments and event reminders to mobile devices via email or via an agent based interface.

RCal can also import schedules marked up in RDF from the Semantic Web and use this markup to reason about the schedule and thus task other agents in the agent system to provide more information related to the schedule.

10. Future Work

RCal is the first intelligent distributed meeting scheduling system that integrates with a commercial PIM and navigates the semantic web thus providing a complete meeting scheduling and calendaring solution for office environments. RCal also provides notifications and runs within an agent society playing an assistive role in human life. RCal can schedule meetings and import calendars from the semantic web and also provide access to user calendars via web based interfaces such as the E-Secretary agent.

However this is only a glimpse of what is possible with such technologies and is a first prototype in this domain of work. The meeting scheduling engine in RCal can only schedule new meetings and is unable to renegotiate meetings that were previously scheduled. This would require improved bookkeeping that would track the appointments and the attendees within the MAS and continue interaction with them even after the meeting has been scheduled. This is different from the current system because RCal treats every negotiation as an atomic interaction with a fixed set of peers. Once a meeting is scheduled there is no mechanism for canceling or changing the meeting time. In addition to this RCal cannot restart failed negotiations with new contracts unless a new instruction to schedule a meeting is issued by the user.

Future version of RCal will also explore different bidding strategies that maximize the effectiveness of the scheduling process. These improvements will also lead to improved techniques of searching for appropriate meeting times rather than the best first strategy currently employed.

The meeting constraint set is currently limited to the time and date of the meeting and does not take into account the location of the meeting or the attendees. While RCal does have facilities for optional attendees there is no provision for changing the status of an attendee while a meeting is being scheduled. Every attendee is treated as a required attendee and must agree to the constraints before the event can be scheduled. Similarly the location of the meeting is also specified as part of the negotiation but there are no facilities for selecting different meeting times based on the availability of a particular location. This can be changed by treating possible location as resources that must be allocated in addition to the time resource currently being negotiated about. The location resource can be added as another constraint to the current set of constraints.

Finally a major component of successful meeting scheduling is managing user preferences. RCal allows a small subset of user preferences to be added and rules that must be obeyed when scheduling meetings. The current version of RCal allows for simple rules such as do not schedule meeting before 8 AM and after 6 PM. The simple preference management module merely sets the times specified by the rules as busy event in the volatile data store so that these times are displayed as busy to the scheduling engine. This is rudimentary at best and does not allow for more complex preferences to be set such as those that take the priority of certain attendees into account and offer earlier meeting times to them.

11. References

1. Ephrati, E., Zlotkin, G. and Rosenchein, J.S., "A Non-Manipulable Meeting Scheduling System", *Proceedings of the 13th International Workshop on Distributed Artificial Intelligence*

2. Garrido, L. and Sycara, K., "Multi-Agent Meeting Scheduling: Preliminary Experimental Results", *In Proceedings of the Second International Conference on Multi Agent Systems (ICMAS-96),* Keihanna Plaza, Kyoto, Japan, Dec., 9-13, 1996.

3. Sycara, K. and Liu, J., "Distributed Meeting Scheduling", *In Sixteenth Annual Conference of the Cognitive Society,* 1994

4. Sen, S. and Durfee, E.H., "A Contracting Model for Flexible Distributed Scheduling", *Annals of Operations Research*, volume 65, pages 195-222, 1996.

5. Sen, S. and Durfee, E.H., "A Formal Study of Distributed Meeting Scheduling", *Group Decision and Negotiation*, volume 7, pages 265-289, 1998.

6. Sen, S. and Durfee, E.H., "On the design of an adaptive meeting scheduler", *In Proceedings pf the Tenth IEEE Conference on Artificial Intelligence for Applications* (pages 40-46), San Antonio, Texas, March 1994.

7. Sen, S., Haynes, T. and Arora, N., "Satisfying User Preferences while Negotiating Meetings", International Journal of Human Computer Studies, vol 47, pp. 407-427, 1997.

8. Shintani, T., Ito, T. and Sycara, K., "Multiple Negotiations among Agents for a Distributed Meeting Scheduler", *In Proceedings of the Fourth International Conference on MultiAgent Systems*, July, 2000, pp. 435 - 436.

9. Garrido, L., Brena, R. and Sycara, K., "Cognitive Modelling and Group Adaptation in Intelligent Multi-Agent Meeting Scheduling", *In Proceedings of the First Iberoamerican Workshop on Distributed Artificial Intelligence and Multi-Agent Systems*, 1996, pp. 55 - 72.

10. T. Berners-Lee, J. Hendler, and O. Lassila, “The Semantic Web,” *Scientific American*., vol. 284, no. 5, May 2001, pp. 34–43.

11. The Resource Description Framework (RDF). http://www.w3c.org/RDF/

12. The iCalendar Specification. http://www.ietf.org/rfc/rfc2445.txt

13. R. G. Smith. "The Contract Net Protocol: High-Level Communications and Control in a Distributed Problem Solver", *IEEE Transactions on Computers*, C29(12), 1980.

14. Mitchell, T., Caruana, R. and Freitag, D., McDermott, J. and Zabowski, D., "Experience with a Learning Personal Agent", *Communications of the ACM*, 37(7):80-91, 1994.

15. IBM Lotus Software. http://www.lotus.com/

16. Web Event. http://www.webevent.com/

17. Netscape Calendar. http://wp.netscape.com/communicator/calendar/v4.0/

18. Yahoo! Calendar. http://calendar.yahoo.com/

19. MSN Calendar. http://calendar.msn.com

20. Microsoft Outlook. http://www.microsoft.com/outlook/

21. Apple iCal. http://www.apple.com/ical/

22. The iCalendar Specification. http://www.ietf.org/rfc/rfc2445.txt

23. The DAML Time Ontology. http://www.cs.rochester.edu/~ferguson/daml/

24. The Hybrid iCal Ontology. http://ilrt.org/discovery/2001/06/schemas/ical-full/hybrid.rdf

25. Sycara, K., Paolucci, M., Van Velsen, M. and Giampapa, J., "The RETSINA MAS Infrastructure", *The special joint issue of Autonomous Agents and Multi-Agent Systems*, Volume 7, Nos. 1 and 2, July, 2003.

26. Shehory, O. and Sycara, K. "The Retsina Communicator". *In Proceedings of Autonomous Agents and Multi-Agent Systems*, 2000.

27. Finin, T., Labrou, Y. and Mayfield, J., "KQML as an agent communication language" *in Software Agents'*, Jeff Bradshaw (Ed.), MIT Press, Cambridge, (1997).