Group Rendezvous in a Synchronous, Collaborative Environment

Jörg Roth, Claus Unger

University of Hagen, Department for Computer Science, 58084 Hagen, Germany
{Joerg.Roth, Claus.Unger}@Fernuni-Hagen.de

Abstract. Before a session in a synchronous, collaborative environment can really start, various actions have to be performed: users have to be informed about planned sessions, network access paths between session participants have to be determined, etc. In the following, we call all these actions *group rendezvous*. In existing groupware systems, the group rendezvous is often neglected, i.e. it is assumed that a team has already formed, session information has been distributed and network paths are known. In reality, this assumption often does not hold: members with dial-up connections are not permanently online, if they get their network addresses from an address pool, e.g. via a network access provider, they are difficult to find even when they are online. This paper describes a fully decentralised group rendezvous system, which addresses these problems. Besides session distribution, the system has a component for resolving variable network addresses. It has been integrated into the synchronous groupware system *DreamTeam* [RU98].

1 Introduction

Synchronous groupware brings together users which are geographically distributed and connected via a network. Many synchronous groupware systems are based upon the session metaphor [RG96] (also known as „groupware as meeting“), where users can join an existing session, can collaborate with other team members and finally can leave a session when their work is done.
To build a session, the date, the place (in terms of network locations), the collaborative environment as well as the session topics have to be specified and distributed; shortly before a session starts, it has further to be determined, who is currently online and how other group members can really be accessed. We call all these operations *group rendezvous*.
[Scho96] defines the rendezvous as the action of inviting other users or scanning for open sessions. This includes Email or WWW-based systems. This kind of rendezvous has a long-term character - inviting someone via Email or posting a meeting date on a bulletin board is usually performed hours or days before the session starts. We call this kind of rendezvous *long-term rendezvous*. In contrast, the last actions, which are to be performed shortly (i.e. a few minutes or even seconds) before a session starts, are called *short-term rendezvous*. In this phase, the group members connect to the

network if necessary (e.g. establish a modem connection) and start their groupware systems which then try to detect other members.
A centralised rendezvous approach, as it is realised in many existing groupware systems, requires a well-known server which holds the states of all group members as well as the session profiles. The realisation of such an approach is easy: a system simply asks the central server (or the *registrar* [RG96b]) for a session list. When a newcomer wants to join, he gets a list of all group members in the session and their network addresses. The registrar on the other hand stores the newcomer's address for further queries. The registrar must be highly reliable, with regard to hardware as well as software. A stopped or failed registrar prevents a session from being started, even if the following communication would run in a decentralised way. Thus, a registrar can be viewed as the weak point of a groupware system.
Our approach is based upon a fully decentralised architecture and provides a solution for both rendezvous phases. The corresponding rendezvous component is an integral part of our own groupware system *DreamTeam* ([RU98], [RU98b]), which will be briefly described in the next chapter.

2 The DreamTeam environment

DreamTeam is a platform for synchronous collaboration and offers a variety of services for application developers as well as for end-users. The DreamTeam environment allows the developer to develop co-operative applications like single user applications, without struggling with network details, synchronisation algorithms, etc. The environment consists of three parts: a development environment, a runtime environment and a simulation environment. The development environment [Roth98] mainly consists of a huge Java class library which contains groupware specific problem solutions as building blocks. The runtime environment provides an infrastructure with special groupware facilities. A front-end on top of the runtime environment allows end-users to control and configure the system. Finally, collaborative applications can be tested in the simulation environment, which allows to simulate network characteristics on a single computer.
DreamTeam is based upon a completely decentralised architecture, thus there is no central server holding session states. The decentralised architecture leads to more complex algorithms, nevertheless performance bottlenecks are avoided and the system is much more reliable. Based on this architecture, we realised a rendezvous component, which will be described in the following in some more detail.

3 Group rendezvous

As mentioned above, we distinguish between long-term rendezvous and short-term rendezvous. As they are based upon different concepts, we will describe them in different chapters.

3.1 Long-term rendezvous

If a team starts organising a session via a synchronous groupware system, several decisions have to be made: regarding the date and duration of the session, the tools to be used during the session, participation restrictions etc. Such a set of information is called a *session profile*.
The DreamTeam session concept allows team members to define session profiles. The user, who starts a session using a session profile, is called the *originator*. Only the originator is allowed to start and stop a session or to modify the session profile. In addition to sessions which are defined by local users, profiles of remote sessions are stored by the groupware system. The list of all available session profiles is called the *session list*, which gives the user an overview about planned sessions.
Once a session has been started, other team members may join. In order to join, the session profile has to be made available locally, i.e. has to be copied to each member's session list *before* the session starts.
Since DreamTeam does not provide a central server, session information must be distributed in a decentralised way, based on standard Internet services. We use the Email and Newsgroup services for session announcements (Figure 1).

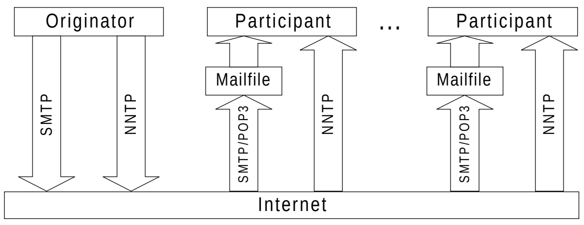


Figure 1: Session announcement mechanism

An originator who has created a session profile and wants to distribute this information to other group members, can either send emails or put an announcement in a predefined newsgroup. For this, the DreamTeam rendezvous component supports SMTP (simple mail transfer protocol [Pos82]) as well as NNTP (network news transfer protocol [KL86]). Announcements can simply be posted within the DreamTeam environment without using external tools (e.g. news or mail reader).
Besides a user defined announcement text, the binary representation of the corresponding session profile is attached to the message. Whenever a DreamTeam environment is started, it scans the corresponding mailbox's incoming file and the newsgroup. When the scan is successful, the announcement message is presented to the user and the binary session information is decoded. Afterwards, the session profile can be included into the local session list, which in turn enables the user to join the corresponding session.

3.2 Short-term rendezvous

While long-term rendezvous support long-term planning and announcements of sessions, at the moment the session actually starts, the user's system must get additional information, which cannot be included in the session profile. This information consists of

- the current list of users who want to attend the session,
- the list of their corresponding network addresses.

Especially network address resolution is not easy in a decentralised system. Since current network addresses are often not assigned before a user goes online, an Email or Newsgroup based approach is too slow. Before presenting our solution to this problem, we introduce a few definitions.

3.2.1 The short-term rendezvous problem in general

Let $H = \{h_1, \ldots h_n\}$ denote the set of hosts in the collaborative group. To every $h_i \in H$ we assign a unique identifier ID_i.

In a decentralised rendezvous, there does not exist a well-known host which holds network addresses or online states. The problem becomes even worse, if there exist hosts with variable network addresses. To identify such hosts inside a network, it is necessary to send messages to all potential addresses and wait for an answer. If no answer returns, with high probability the corresponding host is offline, otherwise the answer includes the correct network address. Each host has to maintain a list of all potential addresses of all other hosts.

Let $adr_{pot}(h)$ denote a set of potential addresses of host *h*. The value of $adr_{pot}(h)$ can be determined:

We assume that a host either permanently belongs to a LAN or can be connected to the network via a dial-up connection (e.g. modem or ISDN). We further assume that a host never changes this characteristic during its lifetime.

In the first case the network address is fix, thus $adr_{pot}(h)$ has only one element. Hosts with dial-up connections get their addresses from a pool of reserved addresses, normally administered by an access provider. Since for each dial-up point this pool is fixed, $adr_{pot}(h)$ can be set to this pool's addresses.

The considerations above lead to a number of algorithms for solving the short-term rendezvous problem. One possibility is to find all other hosts in *H* via multicast. Such an algorithm is described in [Fu98]. Unfortunately, this method leads to an unacceptable high network load. Since every newcomer *h* performs a multicast search for every other host, a total of $\sum_{m \in H \setminus \{h\}} \left| adr_{pot}(m) \right|$ addresses has to be checked. Even the absence of a reply message does not necessarily mean that the corresponding host is really offline. Thus, addresses have to be tested multiple times, which even increases the load of the network.

The following approach significantly reduces the average network load.

3.2.2 The idea

In the following, we introduce a decentralised algorithm, which avoids bulky multicasting and thus reduces the average network load. Multicasting cannot always be avoided, because there may be situations, where no host with a permanent network address is online. The algorithm works as follows:

Every host *h* stores a list L_h of entries $<ID_1,adr_1,t_1>$, $<ID_2,adr_2,t_2>$, ..., $<ID_n,adr_n,t_n>$. We call L_h in the following the *online list* of *h*. Each element stores the actual state of every host h_i. The entries are

- ID_i: the host identifier,
- adr_i: the current network address, or „off“ if the host is offline,
- t_i: the local time host h_i made the last state change (see chapter 3.2.3).

Whenever a group member wants to join a session, his system first builds its own online list. Hereto its rendezvous component queries all known hosts - beginning with permanent addresses. A query is successful, if another host is already online and answers the request. This other host has already built its own online list which it now transfers to the newcomer. The newcomer can now ask all hosts on his actual online list to update their online lists. This procedure ensures that, after a certain delay, all hosts know the state and address of all other hosts in their group. Even in case of variable addresses, a broadcast can be avoided if at least one group member with a permanent address is online. In the worst case (every group members have variable addresses), a broadcast is necessary.

Figure 2 shows a sample session. Every O_i denotes the set of hosts, which host h_i views as online. We assume that host 1 and 2 are already online, host 3 is offline and host 4 is about to start. The first query of host 4 is directed to host 3 which is unable to reply. The second query is directed to host 2 which transfers its own online list. Host 4 now knows that there is a host 1 which has to be informed. After this, the newcomer host 4 as well as all other hosts have correct online lists.

The example continues with host 3 going online and host 1 going offline.

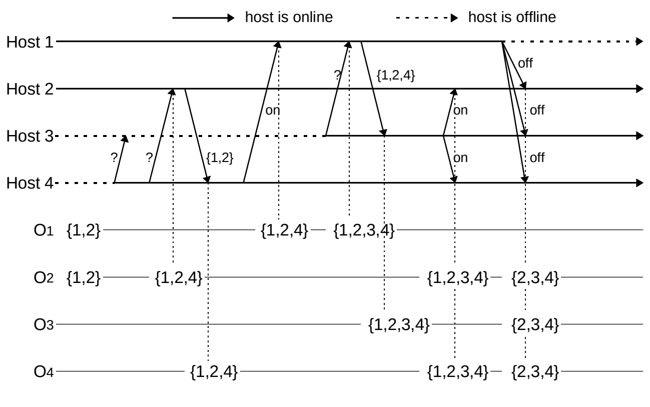


Figure 2: Short-term rendezvous session

In pseudo code, the algorithm for host *h* reads as follows:

```
Main Program:
state:="going online";
build a list Lh and initialise it with <IDi,"off",0> for all hosts $h_i \in H$;
replace element <IDh,adrh,th> in Lh by <IDh,local network address,current time>;
find another host which is in the state "doing work" and get its online list ; // see below
inform all other hosts hi in Lh with adri<>"off" about going online;
state:="doing work";
// perform tasks, e.g. join a session
...
// task terminated
state:="going offline";
replace element <IDh,adrh,th> in Lh by <IDh,"off",current time>;
inform all other hosts hi in Lh with adri<>"off" about going offline;
```

In this algorithm, the find operation is the most complex one and will later be described in more detail. If necessary at all, broadcasting is limited to the find operation, i.e. after a find has been performed, no further message to $adr_{pot}(h)$ is necessary.

3.2.3 Avoiding race conditions

Unfortunately, the algorithm above can cause race conditions which may result in wrong online lists. All race scenarios have in common that state changes are happening during a short period of time. An online list just being transferred from another host may be outdated when it reaches its target.
To address this problem we use the time stamps t_i. Whenever host *h* receives a state change message from *m*, both online lists are compared. If one list contains a newer entry, this entry replaces the older entry in the other list. This operation is called *balance* operation (see below). The following listing describes the complete algorithm, which avoids races:

```
Main Program:
state:="going online";
build list Lh and initialise it with <IDi,"off",0> for all hosts $h_i \in H$;
replace element <IDh,adrh,th> in Lh by <IDh,local network address,current time>;
find another host which is in the state "doing work" and get its online list ; // see below
while (there exists a host i in Lh with adri<>"off" which has not been informed yet)
begin
    send(inform-online, Lh) to i;
    receive(Li);
    balance(Lh, Li);
end
state:="doing work";
// perform tasks, e.g. join a session
...
// task terminated
state:="going offline";
replace element <IDh,adrh,th> in Lh by <IDh,"off",current time>;
```

```
while (there exists a host $i$ in $Lh$ with $adri$<>"off" which has not been informed yet)
begin
    send(inform-offline, $Lh$) to $i$;
    receive($Li$);
    balance($Lh$, $Li$);
end
```

The following thread is being executed, whenever a the host receives a message:

```
**Receive Message:**
receive(message kind, $Lm$);
balance($Lh$, $Lm$);
case message kind:
    inform-online, inform-offline:
        send($Lm$) to $m$;
end
```

The balance operation reads as follows:

```
**balance($Lh$, $Lm$):**
for each element <$IDi$,$adri$,$ti$> in $Lm$ begin
    get element <$IDj$,$adrj$,$tj$> of $Lh$ with $IDi=IDj$;
    if ($ti>tj$)
        replace element <$IDj$,$adrj$,$tj$> in $Lh$ by <$IDi$,$adri$,$ti$>;
    fi
end
```

Note that the list L_m is added to every state change message, host *m* performs. On the other hand, the host *h* has to reply L_h in order to update *m*'s list.
The balance operation ensures, that at least when a newcomer informs all other hosts about going online, the new state information is exchanged. No time synchronisation problems can occur, since t_i and t_j were measured by the same host ($ID_i=ID_j$). Instead of using a real-time clock for setting t_i, a counter can be used as well.

3.2.4 States and abnormal terminations

Figure 3 shows all rendezvous states. The state „offline" is not a program state in the usual sense, since terminated programs have no variables nor can they react on messages.
A normal life cycle runs clockwise through all these states. An abnormal termination leads to consistency problems. A host which terminates abnormally (e.g. because of hardware problems) is no longer able to inform other hosts about its state change. Other hosts would view such a host as online until they try to communicate. A communication request to a terminated host results in a time-out which can be used to correct the state information about this host.

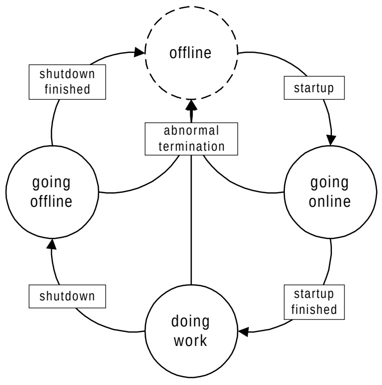


Figure 3: States

In order to find abnormally terminated hosts, even if no communication is in progress, the rendezvous component cyclically sends test messages to each other host, thus ensuring that after a certain time an abnormal termination is detected.

3.2.5 Performing the *find* operation

To avoid unnecessary multicast operations, it is useful to try hosts with permanent addresses first. Another criterion may be the average time, a host was online in the past. The longer a host was online on average, the higher is the probability that he is online when a find operation is performed. We define the average online time as follows:

$$ot\colon H \times IR \times IR \to [0,1],\ ot(h, t_{past}, t_{end}) = \frac{\int_{t_{end}-t_{past}}^{t_{end}} \delta(h,t)\,dt}{t_{past}} \quad \text{where } \delta(h,t) = \begin{cases} 1 \text{ if } h \in O(t) \\ 0 \text{ otherwise} \end{cases}$$

The average online time can only be recorded by a host itself, but can easily be distributed inside the *balance* operation. Since a host can only compute the online time inside an online period, t_{end} cannot be chosen arbitrarily. Thus, we use a further definition:

$$ot_{last}\colon H \times IR \times IR \to [0,1],\ ot_{last}(h, t_{past}, t) = ot(h, t_{past}, \max\{\tau \mid \tau \le t \wedge h \in O(\tau)\})$$

We can now construct a *find* operation:

```
Find:
found:=false;
build a sorted list of all $m \in H \setminus \{h\}$,
    sorted by $|adr_{pot}(m)| \cdot weight_1 - ot_{last}(m, t_{past}, \text{current time}) \cdot weight_2$;
for each host m in the list begin
    send(find-request, $Lh$) as multicast to $adr_{pot}(m)$;
    wait($twait1$);
    if (found) stop; fi
```

```
end
wait(twait2);
```

The receive thread has to be extended as follows:

```
Receive Message:
receive(message kind, Lm,);
balance(Lh, Lm);
case message kind:
    find-request:
        send(find-reply, Lh, state) to m;
    find-reply:
        receive(statem);
        if (statem =“doing work“) found:=true; fi
    inform-online, inform-offline:
        send(Lm) to m;
end
```

Note that the balance operation has also to be executed for message kinds find-request and find-reply in order to avoid race conditions during start-up.
The find operation depends on the following constants which can be tailored to specific environments:

constant	**function**	**sample value**
weight1	weighting the number of potential addresses	1.0
weight2	weighting online time in the past	1.0
tpast	how long in the past should the online time be considered	5 days
twait1	delay after each trial	100 ms
twait2	time-out after which no reply is expected anymore	3 s

Table 1: Parameters for the find operation

From these parameters, *twait1* is the most crucial one. If the value is small, all messages are delivered before the first reply arrives, which causes a high network load but very short response times. On the other hand, if the value is too big, the procedure takes a long time. The optimal value depends on the specific environment.
A high value of *twait2* can cause a newcomer waiting too long, if no other host is online. *twait1*+ *twait2* should be at least as big as the time, a reply from the most distant host needs to cross the network. If both times are to small, potential replies are ignored. In the worst case this can cause wrong online lists.

3.2.6 MBone enhancements

MBone technology [Dee89] offers new possibilities, especially for the rendezvous problem. MBone is a network facility which reduces network load if a sender sends the same message to a group of receivers. Inside an MBone-enabled network, a data package which is directed to many receivers is duplicated at the very last moment,

thus network resources are used economically. It is more efficient to use MBone rather than transmitting the same data package inside a loop, especially in case of bulky real-time transfers e.g. video or audio.
We are interested in a different feature of the MBone technology: to attend a multicast group, a receiver has not to check in centrally. The entire architecture of MBone is decentralised, thus it significantly speeds up our decentralised rendezvous algorithm.
Unfortunately, MBone multicasts are like UDP datagrams [Pos80] and are not suitable for protocols which rely on correct delivery and packet ordering. Whereas our algorithm as a whole is therefore not a good candidate for using MBone, the find operation can benefit from MBone multicast. We modify the find operation as follows:

```
Find:
found:=false;
send a multicast message to a predefined multicast group;
wait(twait1);
if (found) stop; fi
build a sorted list... // rest see above
```

If the newcomer and at least one other host are connected to an MBone-enabled network, the operation is completed immediately, thus no further multicasts are required. Otherwise, only the time *twait1* is wasted. This method uses MBone capabilities whenever they are available. Nevertheless, MBone technology is not a prerequisite for the correct execution of our algorithm.

3.2.7 Discussion of correctness

In the following we assume that busy periods, where several hosts change their states, are followed by sufficiently long idle periods where state information is exchanged between hosts. We will sketch a proof that after a certain time all hosts have identical online lists.
Let us assume that at time t_0 there exists a non-empty set of hosts $L=\{h_i\}$ with correct online lists. Let us further assume that two new hosts *h*' and *h*'' join the network at t_0.
Case 1: both new hosts request online lists from the same $h_i \in L$: either *h*' or *h*'' gets the correct online list and updates all online lists of $L \cup \{h', h''\}$;
Case 2: *h*' requests an online list from h_i, *h*'' requests an online list from h_j; $h_i, h_j \in L$, $i \neq j$; in the worst case, both online lists are incorrect, i.e. don't contain correct information about *h*'' or *h*'. Both hosts, *h*' and *h*'', send their online lists to all hosts of *L*.
Let t'_i and t''_i denote the times when h_i gets online lists from *h*' and *h*''.

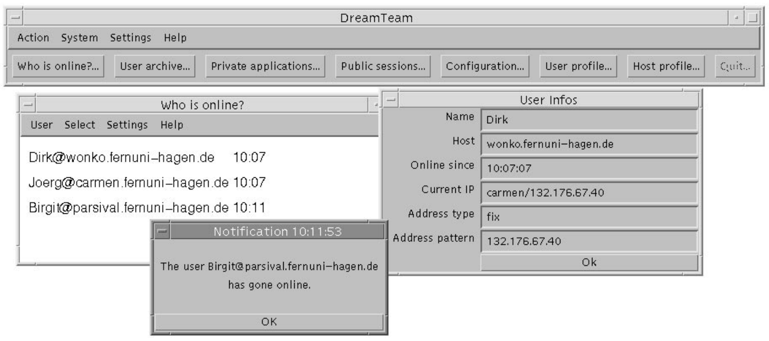
$t''_i < t'_i$: *h*' gets from h_i the correct online list and distributes it to all hosts $L \cup \{h', h''\}$

$t'_i < t''_i$: *h*'' gets from h_i the correct online list and distributes it to all hosts $L \cup \{h', h''\}$.

The proof can easily be extended to the case, where more than two hosts go online at the ‘same’ time. The new host *h** which finally updates the online list of h_i, gets back the correct online list and forwards it to all new and old hosts.
The algorithm even works if no host is already online at t_0. At least one of the new hosts, when vainly searching for a host being in the “doing work“ state, successfully contacts all other new hosts, and, through the balancing operation, correctly updates its and their online lists.

3.3 The front-end

As described above, the rendezvous component is part of the *DreamTeam* collaborative environment. The rendezvous front-end is seamlessly integrated into the DreamTeam front-end. The rendezvous algorithms are implemented as threads and run in the background, more or less independently from the rest of the program. Important state changes open notification windows, which have to be confirmed by the user. Figure 4 shows a typical desktop.



Dirk@wonko.fernuni–hagen.de 10:07
Joerg@carmen.fernuni–hagen.de 10:07
Birgit@parsival.fernuni–hagen.de 10:11

The user Birgit@parsival.fernuni–hagen.de has gone online.

Name	Dirk
Host	wonko.fernuni–hagen.de
Online since	10:07:07
Current IP	carmen/132.176.67.40
Address type	fix
Address pattern	132.176.67.40

Figure 4: The rendezvous front-end

The left window shows the current online list. Besides the system user name („name@host“), the time the user went online is displayed. The right window provides additional information.

4 Related Work

Snapshot problem/distributed termination detection: The distributed termination detection as well as the snapshot problem [CL85] are related to the short-term rendezvous problem. The distributed termination detection finds out, when a distributed computation has been finished or run into a specific state. Snapshot algorithms determine a consistent global state in a distributed computation. Both algorithms have to deal with similar problems as the rendezvous algorithm. Nevertheless, there are two major differences:

- The definition of „state“ in the rendezvous problem includes the offline state which in terms of the algorithms above is not a state at all. A state for the snapshot algorithm as well as for the distributed termination detection means a specific point of computation and applies only to programs which are currently running.
- Both problems assume that all group members can be accessed by a well-known and stable address.

Thus solutions for the problems above cannot be applied to the short-term rendezvous problem.

Registrars: Centralised solutions which we mentioned in the beginning are realised in a variety of session-oriented groupware systems. *Groupkit* and *Habanero* may serve as examples.

Groupkit [RG96b] is based upon a decentralised architecture, the only exception being the group rendezvous. A registrar runs on a well-known server and is the only centralised process required in the Groupkit environment. Whenever a session is created, the session profile is directed to the registrar. A user can load a list of all sessions (running or not) from the registrar and get information for joining. The registrar itself does not handle the creation of sessions, nor is he involved in users joining or leaving sessions. Such requests are relayed to local session managers.

Habanero [NCSA] is fully centralised, thus the group rendezvous is easily integrated into the session management process. In order to enable collaboration, a server application has to run on a well-known server. Once the Habanero server is started, a user can start a client application. Since session profiles reside on the central server, a list of running sessions as well as the network addresses of current participants can be loaded.

Directory services: The session directory service *sdr* [Han96] gives an example for a completely decentralised service. It is based on IP multicast and provides functions for announcing and scheduling sessions (e.g. video conferences). The announcement system can be compared with a radio sender transmitting announcements. The creator of a session distributes session announcements periodically on a multicast channel. Everyone who is interested in announcements has to scan this channel. Whenever an announcement is received, it is added to a local session list. If a receiver fails to receive a specific session announcement for a certain time, the receiver concludes that this session is cancelled and deletes it from the local list.

Besides the announcement system, sdr provides a protocol for distributing session descriptions and scheduling sessions.

Four11 [Four] provides another centralised directory service, the central server is a WWW server which can be accessed via a common web browser. Four11 clients are included into several conference systems such as *CU-SeeMe* [CU] and *Netscape Conference* [Net]. The main idea of the Four11 service is to manage a huge database of registered users. In order to register, users have to submit their names and email addresses. Via this information, it is possible to search for other users manually. In addition, Four11 provides rendezvous support for conference tools. During start-up, the current user is checked into Four11 as being „online“. In order to build sessions, the tool can then retrieve a list of other users which are currently online.

Mobile IP: The *Mobile IP* concept ([Per96], [JP97]) addresses a problem related to the group rendezvous: address resolution. The current IP implementation assumes that a host always resides in the same subnet during lifetime. Since the subnet address is part of the Internet address, a host cannot migrate to another subnet without changing its address. After an address has changed, messages to the old address cannot be delivered any more.
The Mobile IP concept solves this problem. Mobile computers (e.g. laptops) can be connected to the Internet via different subnets without changing their addresses. A so-called *home agent* intercepts incoming messages for the mobile node and sends them to the current address. The sender has not to know about this mechanism. Mobile IP exists as a draft for the actual IP implementation (IPv4) but will be included in the next IP realisation (IPv6).
Comparison: Many existing groupware systems neglect the group rendezvous. Especially long-term rendezvous are often not integrated into the environment and have to be handled manually by an external communication system or tool. We strongly feel that rendezvous support should be integrated into a groupware environment in order to gain acceptance by end-users. The registrar concept is the most often used approach for rendezvous support in existing groupware systems, even if the session communication is decentralised. The registrar is a straight-forward approach for systems which are already organised in a centralised way, but we feel that a decentralised rendezvous much better fits decentralised systems.
The session directory service *sdr* is an example for a decentralised architecture, but covers only session announcements. The lack of persistence mechanisms leads to long online times, both for the sender as well as for a receiver of announcements. In addition, MBone technology is not accessible for a big community, thus a rendezvous system should not solely be based upon MBone. Mobile IP covers only the address resolution problem. Online states as well as session announcements have to be distributed separately.

5 Conclusion

This paper presents a solution for the rendezvous problem. While long-term rendezvous, which emphasises session announcements, short-term rendezvous manages online lists and perform address resolution.
Our solution covers both rendezvous and is based upon a completely decentralised architecture. The long-term rendezvous uses two mechanisms. One mechanism directly exchanges session profiles when two or more members are online at the same time. A second mechanism uses the standard services Email and Newsgroup for distributing session profiles.
The short-time rendezvous is supported by an algorithm which is economically from the view of network traffic. Bulky broadcasts are avoided as far as possible, MBone enabled networks are used to further reduce network loads.

References

[CL85] Chandy K. M., Lamport L.: *Distributed Snapshots: Determining Global States of Distributed Systems*, ACM Transactions on Computer Systems, Vol. 3, No. 1, Feb. 1985, 63-75

[CU] *Enhanced CU-SeeMe Home Page* http://www.cu-seeme.com

[Dee89] Deering S.: *RFC 1112:Host Extensions for IP Multicasting*, Request For Comments, Aug. 1989

[Four] *Four11 directory service*, http://www.four11.com/

[Fu98] Fuchs T.: *Entwurf und Realisierung einer Rendezvous-Komponente für eine dezentrale, synchrone CSCW Umgebung*, Diploma thesis, Fernuniversität Hagen, Apr. 1998

[JP97] Johnson D. B., Perkins C.: *Mobility Support in IPv6*, Mobility Support Working Group, Internet Draft, Nov. 1997

[Han96] Handley M.: *The sdr Session Directory: An Mbone Conference Scheduling and Booking System*, Department of Computer Science, University College London, Apr. 1996

[KL86] Kantor B., Lapsley P.: *RFC 977: Network News Transfer Protocol*, Request For Comments, Feb. 1986

[NCSA] *NCSA Habanero Homepage* http://www.ncsa.uiuc.edu/SDG/Software/Habanero/HabaneroHome.html

[Net] *Netscape Home Page* http://www.netscape.com

[Per96] Perkins C. (ed): *RFC 2002: IP Mobility Support*, Request For Comments, Nov. 1996

[Pos80] Postel J.: *RFC 768: User Datagram Protocol*, Request For Comments, Aug. 1980

[Pos82] Postel J.: *RFC 821: Simple Mail Transfer Protocol*, Request For Comments, Aug. 1982

[RG96] Roseman M., Greenberg S.: *TeamRooms: Network Places for Collaboration*, Proc. of the ACM Conference on Computer Supported Cooperative Work, ACM Press, Nov. 1996, 325-333

[RG96b] Roseman M., Greenberg S.: *Building Real-Time Groupware with GroupKit, A Groupware Toolkit*, ACM Transactions on Computer-Human Interaction, Vol. 3, No. 1, Mar. 1996, 66-106

[Roth98] Roth J., *How to write shared applications with „DreamTeam“*, Technical Reference, Fernuniversität Hagen, Jan. 1998

[RU98] Roth J., Unger C.: *Dream Team - a synchronous CSCW environment for distance education*, Proc. of the ED-MEDIA / ED-TELECOM 98, Freiburg, Jun. 1998

[RU98b] Roth J., Unger C.: *Dream Team - a platform for synchronous collaborative applications*, in Th. Herrmann, K. Just-Hahn (eds): Groupware und organisatorische Innovation (D-CSCW'98), B. G. Teubner Stuttgard 1998, 153-165

[Scho96] Schooler E. M.: *Conferencing and collaborative computing*, Multimedia Systems, Vol. 4, 1996, 210-225