CS400 — Problem Seminar — Fall 2000
Assignment 3: Distributed Calendar Management

Handed out: Sept. 29, 2000
Due: Oct. 13, 2000
TA: Luke Deqing Chen (`lukechen@cs`)

1 Introduction

This fortnight's assignment is mainly in systems, although there are links to theory and AI for those who are interested.

1.1 Distributed Computing

Multiple processors are better than one. One way to improve the throughput of a system is to throw many processors at it and use a parallel algorithm. This is particularly helpful for large problems.

In a distributed system, the available processors may be very numerous, but they are separated by a relatively slow or unreliable network. It is expensive for them to communicate. Fortunately, some problems can be decomposed into relatively independent pieces, so that faraway processors need not communicate much. A famous example is the SETI@home project, which has enlisted millions of idle personal computers in a parallel search for radio signals sent by intelligent extraterrestrials.

This assignment considers another problem where a distributed approach makes sense: calendar management. Other people all over the world might want to see your calendar and pencil things in on it. However, the people who interact most with your calendar—you, and perhaps your colleagues—are probably close in the network to your computer. So storing your calendar on your computer (rather than at some central location) is a good way to reduce communication overhead and vulnerability to network outages. Moreover, this solution distributes the computing load fairly: your computer will devote a lot of cycles and bandwidth to calendar management only if you personally are involved in a lot of meetings.

1.2 Message Passing and Shared Memory

How do processors communicate in a distributed system? Thanks to friendly routers, a machine at one Internet address can send a packet of data to a machine at another Internet address using the Internet Protocol (IP).

Higher-level communication protocols are built on top of IP. The **socket** API provides an interface to two such protocols, UDP and TCP. UDP sockets let a client send a one-time message to a server; such “datagrams” may arrive out of order or not at all. TCP sockets let a client establish a reliable, persistent two-way stream connection with the server, sort of like a telephone call. In both cases, the client initiates contact with a numbered port at an IP address; an appropriate server program must be listening to that port.

It is possible to “stack” even higher-level protocols on top of sockets on top of IP. Most distributed systems invent their own application-specific protocols that let processes send specially formatted messages to each other. For example, the World-Wide Web uses HTTP; email uses SMTP; and SETI@home uses an auditing protocol built on top of HTTP. Other well-known protocols are FTP, telnet, and RCP. A distributed calendar manager’s protocol might involve messages that mean “Please tell me within 10 seconds whether you are free at <time> on <date>.”

In this assignment, we will be using a new general-purpose high-level protocol, InterWeave, that is under development here at URCS. The designers of InterWeave hope to make distributed systems easier to write and maintain. Processes need not send specially designed messages to exchange data. They simply modify variables in **shared memory**. These changes are visible to any other process that cares to look at the shared memory. For example, if part of your calendar is in a shared memory segment, then any process can look at it and (after obtaining a write lock) add appointments to it.

So roughly speaking, InterWeave tries to emulate a CREW[1] parallel computer.[2] Of course the processors do not really share memory. They each keep local copies of the shared memory segments, and use message passing underlyingly to keep these local

[1] CREW = Concurrent Read, Exclusive Write.

[2] Or a random-access filesystem shared by many processes. Some distributed versions of the Unix filesystem, notably Andrew and its successor Coda (but not NFS), resemble InterWeave in that they support relaxed consistency and use slow networks efficiently. However, InterWeave currently has significant semantic differences from Unix filesystems: e.g., no access permissions, different locking semantics, and a wider choice of consistency semantics.

copies up to date. The details of this underlying message passing protocol are handled by InterWeave, and remain transparent to the applications built on top of them. But to reduce the underlying communication burden, InterWeave uses a “relaxed consistency” model: memory segments can specify that they are allowed to be a certain amount out of date. This model is part of the semantics of InterWeave, i.e., it *is* visible to and manipulable by application programs.

To get a sense of the wide range of interest and issues in current Distributed Shared Memory (DSM) research, you might check out `http://www.cs.utexas.edu/users/kistler/cs395_dsm/dsm_links.htm`.

2 Assignment Overview

This assignment is intended to give you practice in designing, building, and debugging a concurrent system. You will also be exposed to InterWeave and perhaps other tools. In fact, since you are among the very first users of InterWeave, the systems group is eagerly looking forward to your reports, and is hoping to use your work as a demo!

Even using InterWeave, you will still need to design something like an application-specific protocol for calendar management. But instead of agreeing on the format and sequence of messages, the different processes will have to agree on the format of shared memory and the conditions under which it is appropriate for a process to modify a memory location. The InterWeave team hopes that this kind of design task is easier for programmers..

Distributed systems have to be designed carefully, so that deadlocks or local failures don’t bring the whole system down. (See an OS text such as *Operating System Concepts*, by Silberschatz and Galvin, for more information.) Discussion helps avoid bugs. Because of this, and because there’s more code to write than last time, you should approach the task in 3-person teams.

Here’s what each team should do:

1. Design and implement a simple distributed calendar manager, starting with the minimal design requirements below.

2. Extend this simple project in some way. Some recommended extensions are given below; check with me if you want to work on something else.

3. Write reports. As usual, each team member should write his or her own report.

I’ll probably try to arrange equipment for in-class demos on the due date.

3 Minimal Design Requirements

3.1 Functionality

Logically, each user's calendar is something like a set of appointments, each of which is a (event description, time slot, commitment level) tuple.[3] The actual data structure and its layout over one or more segments are up to you. You should make it relatively efficient to add appointments, modify appointments, look up the appointments that overlap a given time slot, etc.

The **event description** might simply be a unique text string such as "grant meeting" or "doctor's appointment." Or it might be a pointer to a shared record containing various information about the event (e.g., the number of currently confirmed participants)

Time slots could have arbitrary start and end times, or you may want to divide each day into fixed half-hour slots. You might allow a special kind of time slot for recurring events (e.g., the CIS colloquium is every Monday 11-12 during a certain range of dates); or you might choose to handle recurring events with multiple ordinary time slots.

The **commitment level** is used for collaborative scheduling of events. A user of the calendar manager should be able to organize a group meeting as follows. This is not an automated procedure: every step involves a human decision.

1. Construct a list of users to invite.
2. Look at the invitees' calendars and pick a time slot that might work for most of the invitees.
3. Write the meeting "in pencil" at this time on all the invitees' calendars. (Do this even if some of the invitees currently have conflicting commitments at that time—they might change those commitments!)
4. Wait for most of the invitees to accept or decline the proposal. They will do this by writing "yes" or "no" next to the proposal on their calendars, indicating whether they would be willing to attend at that time. They may change their decision at any time.
5. Monitor the invitees' responses. After enough of the invitees have answered (or enough time has elapsed), decide whether this time will work. If so, convert the pencil meeting into ink on all the calendars. If not, erase it from all the calendars and start again.

[3] Or, to take an even broader view, a "calendar space" is a set of (user, event description, time slot, commitment level) tuples.

Some remarks:

- A benefit of shared memory is that the meeting organizer can see the status of all other proposals on the invitees' calendars. This makes it easier to pick a promising time. Shared memory also allows the meeting organizer to modify the invitees' calendars.

- A deficiency of shared memory (in InterWeave) is that anyone can write anywhere. Hence the above protocol relies on conventions, not enforceable mechanisms. A malicious or buggy process could improperly modify the calendars. (This will be partly fixed when permissions are implemented for InterWeave, so that malicious or buggy processes do not pose a threat unless they have write permission on a segment!)

- The meeting organizer might propose several time slots simultaneously, by penciling in the same event at several different times. After invitees have indicated the times at which they would be willing to attend, the organizer inks in the best time and erases the others.

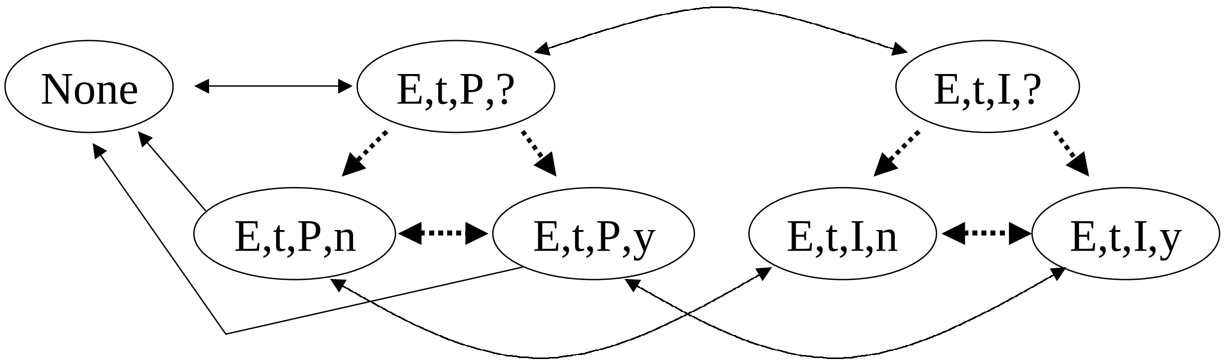
- The system might prevent a user from marking two overlapping events with "yes." More flexibly, it might allow this behavior but warn the user visually or with a message.

- Once a meeting has been scheduled in ink, invitees can continue to change their "yes" or "no" status.

- After a meeting has been scheduled in ink, the organizer can still delete it or convert it back to pencil.

Thus, the commitment level of an appointment is a pair in $\{pencil, ink\} \times \{unknown, yes, no\}$. The first component indicates the meeting organizer's commitment to holding the meeting at this time; it should only be modified by the meeting organizer. The second component indicates the invitee's commitment to attending if the meeting is held at this time; it should only be modified by the invitee.[4] The state diagram is therefore as follows, where solid lines are organizer moves and dotted lines are invitee moves:

[4] For greater sophistication, you could use more finely graded commitment levels, e.g., probability pairs in $[0, 1] \times [0, 1]$.



(A few additional moves are allowed by transitivity; some or all of these could be allowed as atomic moves as well, but I left them out to simplify the diagram.)

Often people simply mark dates on their own calendars. In this case, user is both the organizer and the only invitee, and has complete control over the commitment level. The initial commitment level should probably default to (ink, yes) rather than $(pencil, unknown)$ in this case, although other commitment levels sometimes make sense.

3.2 Privacy and Security

Privacy usually requires limiting read access. Security usually also requires limiting write access, so that processes cannot corrupt data or hold writer locks indefinitely.

At a minimum, your implementation should allow private event descriptions. For example, Nancy might put "oncologist's appointment" on her own calendar. Other users should be able to see that she has an event in that slot, but not what the event is.

Notice that Nancy's private event descriptions might be stored in some process's unshared memory, or in a shared memory segment whose URL is not disclosed to other processes, or in a shared memory segment in encrypted form, or in a file that only Nancy's processes can access.

This is not very sophisticated privacy. For example, only individuals can keep their events private, not groups like Alcoholics Anonymous. Can you think of any other privacy threats raised by this simple calendar management system?

Your report should say something about privacy and security issues. What are the issues for this application? Briefly, how would you extend the design of InterWeave to let you deal with these? Could you deal with any of them under the current InterWeave design (e.g., using cryptography)?

3.3 Interface

In keeping with Unix design philosophy, you should build a simple command-line interface. This is straightforward to design and implement. It is also easy to document, extend, and debug. Finally, it is flexible because other programs can easily send text commands to the calendar manager:

- A GUI (which you might build later) can send commands in response to mouse actions.
- The input can come from a file; this is useful when trying to replicate a run during debugging.
- You can automate common tasks by packaging a sequence of several commands into a script.
- A randomized script can test the program by sending it long sequences of commands (e.g., simulating a random user).

You will need to design a small command language. You might include commands similar to these:

```
LOGIN goldman heart.cs.rochester.edu/calendar/emma
NEWEVENT e = "anarchist’s club meeting"
FINDEVENT e = "anarch"
TIME t = Sep 30 2000 08:45-11:15
GROUP a = proudhon bellegarrigue brousse malatesta merlino goldman
          berkman kropotkin
PENCIL e t a
INK e t a
ERASE e t a
ACCEPT e t
DECLINE e t
ADDORGANIZER e berkman
QUITORGANIZER e
PRIVATE e
BACKUPTO goldman.cal
RESTOREFROM goldman.cal
SOURCE command_file
```

Of course, not all commands are valid in all circumstances.

You may also want some commands for querying the calendar and the status of meetings. However, one thing that a command-line interface cannot easily provide is a continuous view of a shared-memory calendar as it changes. So you should write another client program that serves as a viewer. The viewer should be able to display one or more users' calendars in a nice format, and it should update the view periodically as the calendars change.

3.4 Stress Testing

Be sure to test your system under stress conditions where many users are scheduling at once. To do this, write a script (e.g., in Perl) that generates a long sequence of random commands. Start up several copies of the command-line interface and pipe (different) random commands into them at the same time.

This will make InterWeave work hard! In your report, you should comment on the efficiency of InterWeave and your design under the stress conditions. Describe how you conducted the stress tests. Feel free to include empirical measurements of throughput, process blocking. etc.

Stress testing will also help you find any bugs in your code, especially if you have put assertions in the code, which is always wise.

3.5 Hints

- You will have to write at least part of the system in C or C++, in order to call the InterWeave library functions.

- At present, InterWeave only works on Sun machines. Also, the URCS firewall is not configured to let InterWeave traffic through, so you can't share memory between Suns on opposite sides of the firewall.

- Don't hardcode the server names. Your code should run anywhere, not just on iw.cs.rochester.edu. In particular, each user should be able to choose a nearby server where his or her calendar will be stored. (A server is simply any machine running an InterWeave server process.)

- Consider how users will find each other's calendars. Do they need to know the URLs of each other's calendar segments? Is there a directory of such URLs that they can consult?

- When deciding how to divide a data structure across segments, keep the following in mind:

  - Write locks are per segment. More segments allows more simultaneous writers (one per segment). For example, process 1 could add a Monday appointment while process 2 is adding a Tuesday appointment, if the two days are stored in separate segments.

  - There is substantial overhead per segment. It is slower to get read locks on a few small segments than on one large segment. (Synchronizing the large segment might seem to require copying more data, but InterWeave only transmits the portions of the segment that have changed. Even in the worst case where the entire large segment has changed, it is fairly cheap to send a single datagram of up to about 64 KBytes.)

  - Consistency is per segment. A process could end up with simultaneous read locks on version 4 of segment A and version 5 of segment B, so that the data structure formed by the two segments is incoherent. InterWeave does not yet provide a mechanism to demand cross-segment consistency. (However, it is possible to ensure that the version of segment B is at least as recent as the version of segment A, e.g., by acquiring the read lock on A first and then acquiring the read lock on B with a `FULL_COHERENCE` model.)

- Be warned that you *may not store* ordinary machine-dependent pointers in shared memory. They will mean nothing to the other machines sharing that memory.[5] Instead of ordinary pointers, you can store machine-independent pointers (`mip`) or locations (`loc`).

- While you have a read or write lock on a segment, you can use ordinary pointers into the segment to do your processing. However, once you release the lock, these ordinary pointers become invalid: the segment may move to a new memory address. Only machine-independent pointers (`mip`) or locations (`loc`) or remain valid between locking events. Once you get a new lock, you can convert them to ordinary pointers again.

- Future versions of InterWeave will be strongly typed (to allow automatic conversion of data formats and pointers across machines). In particular, the exact type of a memory block returned by `IW_malloc` will have to be specified at compile time.

[5] InterWeave will eventually allow you to use machine-dependent pointers rather freely: it will automatically fix them up in the other machines' copies. However, this isn't implemented yet.

Thus you'll be able to malloc an array of 15 struct foos, but not an untyped block that you can carve up arbitrarily as needed at runtime. Try to respect this future restriction (which should be easy).

- Check the return values of all functions so that you can recover or exit gracefully if an error condition arises.
- Prove that the system will not become deadlocked.

4 Possible Extensions

The systems faculty would strongly like at least one team to pursue the first extension below. Every team must work on some extension; it would be nice if each team did something different.

4.1 Extension: Did InterWeave Help?

As discussed in the introduction, InterWeave's shared-memory paradigm is supposed to be easier than message passing while providing comparable speed and almost as much flexibility.

Is this true or not? Write another version of your system in which the processes communicate using TCP/IP sockets rather than InterWeave. In your report, explain the two designs and compare their elegance, efficiency, and flexibility.

4.2 Extension: Better Interface

The InterWeave team would like to get a nice demo out of this project. Build a graphical user interface (GUI). Add some features that would convince people to actually use this calendar manager in practice.

The GUI design and additional features are up to you. A few ideas to get your started:

- Use the mouse to drag events around, accept/decline invitations, etc.
- Visual devices (e.g., color) should indicate commitment levels, conflicts, and time slots that are waiting for the user to respond.
- Events that a user does not plan to attend should be displayed less prominently, or not at all.

- Notify the user of certain relevant circumstances: someone has invited him to a meeting, someone has canceled a meeting that he planed to attend, everyone has responded to an invitation that he issued, someone has failed to respond to the invitation, etc. The user could be notified by visual events, audio events, or email.
- More support for event organizers: automatically find free times, automatically ink meetings in if enough people accept, etc.

4.3 Extension: Automated Scheduling

For difficult scheduling problems, some automated assistance is needed. A good example is scheduling high school or university classes. If Professor Plum reschedules his class, certain very interested students will switch into it, which frees up other classes to move, etc. Our calendar manager design does allow Professor Plum to switch, but he has no way of realizing that this will improve everyone's happiness in the long run.

The problem can be formulated as a weighted constraint satisfaction problem. We introduce boolean variables with names of the form $attend(U, E)$ and $sched(E, T)$, where U is a user, E is an event, and T is a time slot. If U really wants to attend E, then we add a highly-weighted constraint that $attend(U, E)$ should be true. But we cannot make everything true: there are some extremely highly-weighted constraints that prevent a student from being in two places at once and prevent an event from occupying two timeslots.[6]

The weighted MAX SAT problem is to assign true or false to each Boolean variable so as to maximize the *total weight* of the satisfied constraints. Finding an optimal solution is intractable (recall that SAT is NP-complete), but there are reasonable approximation algorithms. Some of these algorithms lend themselves to a parallel distributed shared-memory implementation.

For this extension, add automated scheduling to the calendar agent. An organizer can ask that an event be automatically scheduled. The automated scheduling of multiple events takes place in parallel, and in parallel with manual scheduling. For example, manually inking an event onto someone's calendar may cause automatic events to be rescheduled around it.

When an event is to be automatically scheduled, the organizer does not pencil it into a time slot on an invitee's calendar, but only specifies how important it is that the invitee come (perhaps 0). The invitee might change that weight at any time![7]

[6] If we want to allow the same event to be scheduled at multiple times—e.g., different sections of a class—there are a couple of ways to modify the setup.

[7] Of course, there needs to be some mechanism to prevent everyone from trying to use higher weights

There are at least two approaches you might consider for automated scheduling with InterWeave:

- Multiple altruistic processes. Each process looks for schedule changes that will improve the global welfare, and carries them out. (Some processes might be devoted to modifying the variables that affect a particular user, event, or time.) Combining this with a measure of randomness can produce very good solutions. The best place to start is probably Selman, Kautz, & Cohen's WalkSAT algorithm. You can find the WalkSAT paper and related information on the web.

- Multiple selfish processes. Each process is an **agent** that represents a user or event organizer. In a market system, each user agent has a fixed amount of money, and can bribe an event organizer's agent to reschedule an event. If the latter agent is offered enough bribes, it may reschedule (but if it reschedules again, it has to refund the bribes). This approach requires rules for the market, and it requires agents that can strategize a little bit about where their money is best spent. The shared memory is used to manage open bidding and bank accounts. Ask me for more details if you are interested in this approach.

Either of these approaches can probably withstand rather relaxed coherence, especially on large problems. You could experiment with different coherence models. You could also experiment with different strategies for addressing the fact that by the time a process has decided what to do, the world may have changed. What gets the fastest convergence to a very good solution?

If you want to be really ambitious, you can allow specification of more general constraints. For example, if U would rather not take CS444 at 9am, U might add the constraint $\neg attend(U, CS444) \vee \neg sched(CS444, \text{9am})$. Additional constraints might limit courseload, teaching load, and enrollment, etc.; allow sufficient time to travel between classes; prefer certain rooms or equipment; etc.

At any rate, you should describe the algorithms and analyze their empirical performance, particularly under stress testing and different coherence models. Because the changes to shared memory are rapid, automatic, and principled (not random) under this approach, this is a very good practical test of InterWeave, as well as being an interesting investigation into randomized parallel algorithms and some newfangled branches of AI.

than everyone else. For example, each user has a limited total amount of weight to throw wherever he or she wants.

5 Resources

5.1 InterWeave

You can find papers about InterWeave at `http://www.cs.rochester.edu/u/scott/interweave/home.html`.

There are man pages for the InterWeave library calls. To access them, you will need to add the following line to your `.cshrc` file:

```
setenv MANPATH /s17/cashmere/iw/man:$MANPATH
```

Use `man InterWeave` for an overview of the library. The SEE ALSO section of this man page lists the other library calls, which have their own man pages.

Luke Chen has written a simple example InterWeave application for you. You can find it in `/s17/cashmere/iw/src/app/chat/`.

The slides from Luke's talk in class are on his homepage, .

5.2 User Interface

When building the command-line interface, I strongly recommend that you use the nice GNU readline facility to read commands from the input. This is easy to do. It makes your program much easier to use, because the user will be able to edit commands and recall past commands. To show the use of readline, I've written a sample program and a couple of useful auxiliary functions for you: you can find them in `/u/jason/teach/cs400/hw3/readline/`.

Your command-line language might be very simple. But if you want something more complicated or powerful, there are various tools that can help you parse and interpret commands. The classical tools are `lex` and `yacc`. Another option is to use the powerful Tcl scripting language, which is designed as a kind of universal command-line interface: you can extend it with new functions implemented in C. Yet another option is to write each command as a separate mini-program and use the shell as your command line. Or you could glue readline to Perl to C. And so forth.

There are many tools that can help you build a GUI. Tk (especially in conjunction with Tcl) is many people's favorite way to get a nice GUI up and running quickly. There are also other GUI toolkits such as Qt and Motif.

If you are just writing a non-interactive viewer, you may not need full GUI power. (You could even write a text-based viewer that runs within an xterm, using the `curses` library .) One easy kludge is for the program to keep updating an HTML file that can be viewed with a browser. If the HTML file includes the tag

```
<META HTTP-EQUIV="REFRESH" CONTENT=3>
```

in the header (between `<HEAD>` and `</HEAD>`), then the browser will sync with the file every 3 seconds.

5.3 Message Passing

Michael Scott has kindly volunteered his copy of *The Pocket Guide to TCP/IP Sockets*, by Michael J. Donahoo and Kenneth L. Calvert (Morgan Kaufmann, 2001). See me if you want it.

6 Reports

Your report should describe and justify the design choices you made. It should especially discuss your choice of consistency model for each segment—when do different models make sense?—as well as other systems issues such as protocols, deadlocks, and performance.

Second, you should report specifically on the extensions your team built. What you say here depends on the particular extension. In general, focus on the aspects of your project that a reader will find most interesting.

Third, as noted above, you should also discuss the performance of your design and of InterWeave under stress conditions.

Finally, comment on the design and usability of InterWeave. Did you find it congenial? How would you improve the API? What new features would help? What are the privacy and security issues and how would you solve them? The InterWeave team is very interested to hear what you have to say!

As always, write your reports individually, even though you worked as a team to build and study the system. And as always, start writing early!