A DISTRIBUTED EDUCATION ENVIRONMENT BASED ON MATHEMATICA

Wolfgang Schreiner*

Abstract

This paper presents the concept of an environment for distributed mathematical education based on the computer algebra system Mathematica©. *Mathematica is a "system for doing mathematics by computer" which combines a computational engine with powerful presentation and visualization facilities based on the concept of* notebooks *(multimedia hypertext documents). The system has gained large popularity in various scientific and technical fields and is also heavily used in education. We have designed a Mathematica-based distributed environment that supports classroom scenarios such as the lecturer presenting a notebook to an audience or the students exercising independently on personal notebook copies. This design demonstrates various aspects of the adaption of commercial software to distributed applications, in particular compromises that arise from the limitations of external control facilities. A prototype of this environment is being implemented in the frame of a diploma thesis.*

1 Introduction

Computer-supported training and education is one of the most promising application areas of the multimedia-augmented computing technologies that have emerged during the last years. A prominent example is the computer algebra system Mathematica© [11] which has become a popular component of modern scientific education, not only because of its facilities for solving complex mathematical problems but also for its sophisticated support for typesetting, presentation, and visualization [3]. Mathematica allows to organize education material in form of *notebooks*, hypertext documents with executable components, which may serve as the basis of classroom presentations as well as of interactive exercises and of work assignments.

Apart from the apparent pedagogical benefits, computer-supported education has another dimension which arises from the more and more advancing integration of computing and communicating technologies: computers may serve as the enabling devices of education scenarios that overcome the limitations of physical proximity and lead to distributed ("virtual") classrooms. While these prospects are very appealing with a large number of potential applications [10],

*Research Institute for Symbolic Computation (RISC-Linz), Johannes Kepler University, A-4040 Linz, Austria; email: Wolfgang.Schreiner@risc.uni-linz.ac.at; URL: http://www.risc.uni-linz.ac.at/people/schreine.

many approaches to tele-learning and tele-education still concentrate on a straight-forward extension of the traditional classroom model by broadcasting lectures with the help of conventional video-conferencing systems.

In our opinion, this focus on video is unfortunate, because it tends to overshadow the importance of other communication and cooperation technologies [4]. Actually, a short analysis reveals that in modern computer-supported education most of the information content of a lecture is *not* provided by video: verbal communication requires high quality audio connections (which may be even harder to establish over wide area networks than decent video support [1]) while non-verbal communication is nowadays more and more based on digital sources (electronic slide shows and computer-aided demonstrations).

There exist various approaches to provide integrated environments for distance learning that take these new facilities into account and combine video/audio technologies with shared software tools for distributed presentation and collaboration [8]. However, a major problem of such systems is how to integrate existing commercial software packages developed for particular science/education domains into the respective distributed frameworks.

In this paper, we present the design of a distributed environment for mathematical education based on Mathematica. This environment is not intended to compete with general purpose distance learning systems; instead it should be viewed as a case study that demonstrates how a special purpose application package can serve as a component of distributed education sessions considering the constraints of commercial software (limited control interfaces). Nevertheless we believe that this design already covers many important usage scenarios; in combination with public domain video/audio support based on Mbone [7] it may itself already present an interesting tool for practical purposes.

The rest of the paper is structured as follows: Section 2 gives a short overview on Mathematica and its essential building blocks. Section 3 sketches the overall design of the distributed environment. Section 4 deals with the problems that arise in the interaction with the Mathematica notebook frontend. Section 5 outlines the protocols handling the various operation modes of a distributed education session. Section 6 describes the multicasting communication interface in somewhat larger detail. Section 7 concludes the paper with an outline of our further plans.

2 Mathematica

The Mathematica environment consists of three major building blocks [11]:

Kernel. The Mathematica kernel is basically an interpreter for the Mathematica programming language. This language combines features from procedural, functional, and rule-based

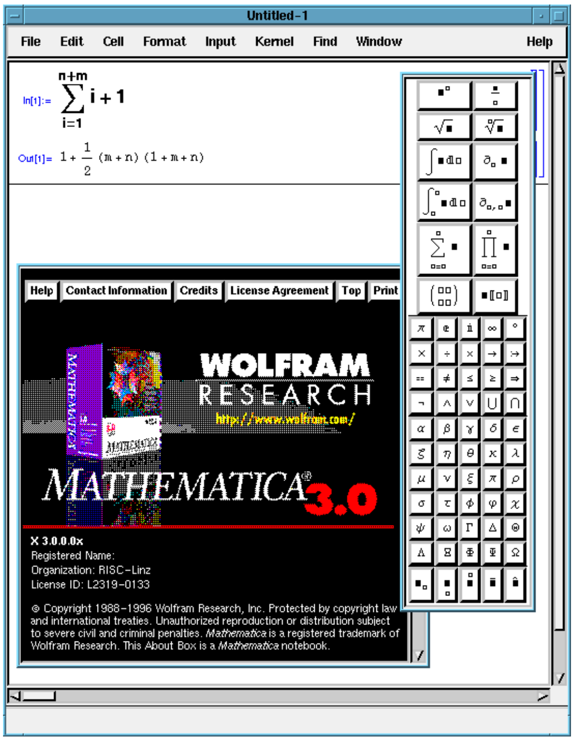

Figure 1: Notebook Frontend

programming; its execution model is essentially based on the concept of *higher-order equational logic* [2]. The kernel has a simple text-based interface by which the user can enter commands and receives the results in textual form, e.g.

```
In[1]:= Sum[i+1, i, 0, n+m]

                         (m + n) (1 + m + n)
Out[1]= 1 + m + n + -------------------
                                 2
```

Library. The Mathematica library incorporates the system's "expert knowledge"; it consists of a large number of mathematical algorithms written in the Mathematica language and interpreted by the Mathematica kernel. Apart from the basic library, many Mathematica packages for various application domains are available. However, because of the poor performance of the interpreted Mathematica language, a couple of time-critical algorithms are directly implemented in the kernel (written in C and compiled to machine code).

Frontend. The Mathematica frontend represents a graphical user-interface to the kernel. Actually, there exist various frontends for different purposes, e.g. for the visualization of computational results. The *notebook frontend* (see Figure 1) allows to write, display, and

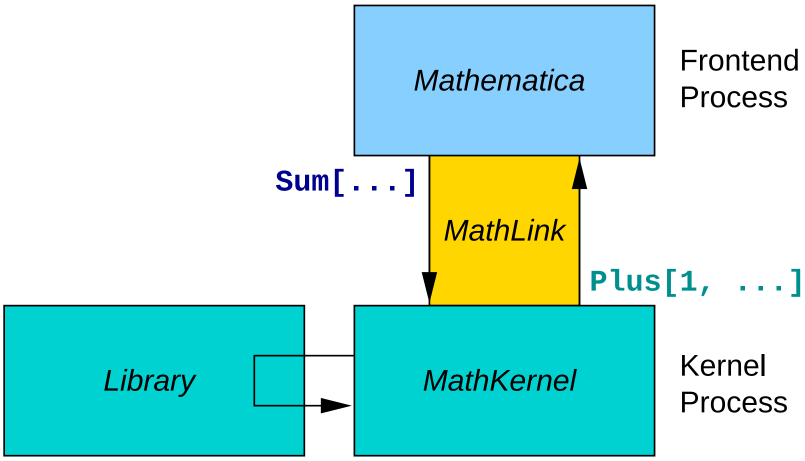

Figure 2: Mathematica Communication

interact with Mathematica *notebooks*, i.e., hypertext documents structured as hierarchies of nested *cells* which may be individually opened and closed. A notebook may embed *active contents*, i.e., Mathematica commands that are executed by the kernel and whose results are displayed inline. By sophisticated facilities for mathematical type-setting and for multimedia components, notebooks serve as the main means of presentation and authoring in Mathematica.

Kernel and frontend are separate processes; they communicate by an external communication mechanism called *MathLink* designed for the portable exchange of Mathematica data (see Figure 2). Thus one may call external programs from within Mathematica or, vice versa, use Mathematica as a component of third-party applications. Moreover, this gives the possibility to run Mathematica kernel and frontend on different machines, to set up alternative frontends, or to exchange data between concurrent instances of Mathematica. The MathLink Developer's kit provides a C library for the development of programs that can exchange data with Mathematica via communication links built on top of the Unix socket mechanism.

A typical scenario for the use of Mathematica in scientific/technical education are lectures based on the presentation of Mathematica notebooks analogous to lectures supported by transparency slides: The lecturer loads a notebook on her own machine whose display is projected to an overhead screen. She scrolls through the document, changes the level of detail by opening and closing cells, and triggers the execution of active contents; the audience listens to the presentation and watches the screen.

Provided that the audience is equipped with own machines, the participants may then load the notebook via some file transfer protocol or network file system into their own instances of Mathematica and independently run exercises, work on assignments, develop own programs,

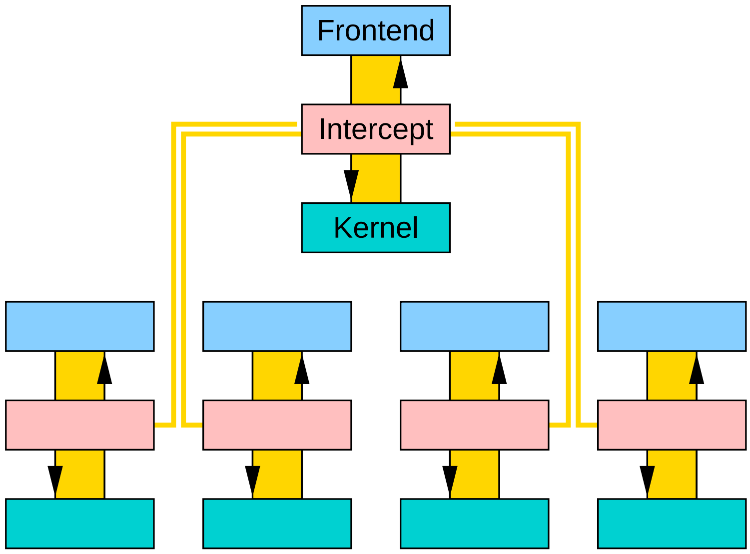

Figure 3: A Distributed Mathematica Environment

and so on. In a similar way, results may be transferred back to the lecturer's host.

However, the use of Mathematica in this scenario is quite restricted and rather uncomfortable:

- By its reliance on optical communication facilities, the presentation is restricted to a local audience. Furthermore, the quality of projection (with respect to brightness and contrast) is typically poor compared to computer displays.
- The coordination of independent exercise modes (involving file transfer, system startup, etc.) and the subsequent presentation of results by students is difficult.

The following sections sketch a system that addresses these problems by integrating the individual instances of Mathematica on different machines into a unified distributed environment.

3 A Distributed Mathematica Environment

Our basic idea of integrating Mathematica into a distributed education environment is to insert a process between frontend and kernel that intercepts their communication and communicates with other interceptor processes in order to implement a distributed session (see Figure 3). This model assumes that each session participant has an instance of Mathematica running (although in some scenarios only a notebook frontend is sufficient) whose interceptor process communicates with the interceptor process of the lecturer. We deliberately concentrate on a star-shaped communication structure, on the one hand to avoid the difficulties of a fully

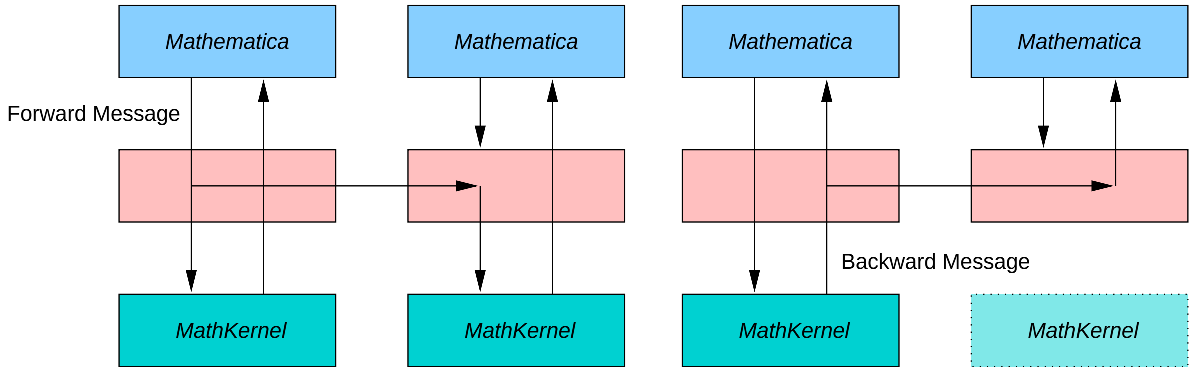

Figure 4: Forward and Backward Interception

distributed system design, on the other hand because it already serves the purposes of the envisioned usage scenarios. This is however not a principal restriction; on demand we may employ more general communication patterns.

There exist two basic possibilities how this interception may be utilized (see Figure 4):

- In *forward interception*, messages from one frontend are transmitted to all kernels; thus one frontend controls the behavior of all kernels. All messages from a kernel are propagated to the respective frontend; thus all frontends display the same contents.
- In *backward interception*, messages from one kernel are transmitted to all frontends; thus the results of one computation are displayed in all frontends.

In both cases, only the messages from one frontend are forwarded to the respective kernel; messages from all other frontends are blocked by the respective interceptors.

Both alternatives have their advantages: forward interception allows to put multiple kernels into a single state from which independent execution becomes possible; backward interception only requires a single kernel and thus avoids inconsistency problems. The protocols sketched in the following sections represent appropriately adopted combinations of these possibilities.

4 Notebook Control

Our original motivation stems from the desire to have the notebook frontend of the lecturer control all the notebook frontends of her audience. For this purpose, two requirements must be fulfilled:

1. The interaction of the lecturer with her frontend must trigger corresponding messages to the kernel (such that these messages can be intercepted).

2. It must be possible to trigger state changes of a notebook in a frontend by external sources (e.g. by messages from the kernel).

Unfortunately, the standard Mathematica frontend fulfills the first demand only to a small extent:

- First, all *user interface events* (like scrolling the notebook in a window) are directly handled by the frontend.
- However, also *basic notebook manipulation* takes place in the frontend; e.g. reading a notebook from a file, opening/closing a notebook cell, selecting and updating the content of a cell, and navigating from one cell to another.
- Only the evaluation of active notebook contents (triggered by pressing the `Shift-Return` key) creates an `EvaluatePacket` sent to the kernel which responds to the frontend by a corresponding `ReturnPacket`.

After so many bad news, it is relieving to see that at least the second demand is essentially met:

- Each notebook open in the frontend may be represented by a a `NotebookObject` in the kernel.
- The kernel may trigger the transfer of the current notebook state from frontend to kernel (command `NotebookGet`) and vice versa (`NotebookPut`). It may also determine the current selection in a notebook in the frontend (`NotebookSelection`).
- Most important, each frontend operation on a notebook may be performed by a corresponding kernel operation.

Consequently, each user event that triggers a change in the state of a notebook can be in principle mirrored by a corresponding operation in the kernel which updates the notebook in the frontend. More specifically:

- A notebook cell may be opened or closed (command `SetOptions`).
- Selected contents may be retrieved (`NotebookRead`) and updated (`NotebookWrite`).
- The current notebook selection in the frontend may be moved from one cell to another (`SelectionMove`, `NotebookLocate`).

- The kernel may trigger the execution of some frontend operations (`FrontEndExecute`).

Of course, this still leaves the problem how to determine the control operations performed by the lecturer, for the frontend does not propagate this information to the outside. Since it is out of question to write a complete notebook frontend from scratch, we see three possibilities:

Embedded Control. We may embed control into a notebook itself by adding *control buttons* to the notebook. Such control buttons are a special form of active notebook contents: when pressed, execution of a user-defined command is triggered in the kernel. By attaching a `SelectionMove` command to such a button, it may serve the same purpose as a link in hypertext documents.

The advantage of this solution is its technical simplicity, but it requires the corresponding modification of notebooks for distributed presentation.

Control Palette. Instead of hardwiring navigation into the document, corresponding facilities may be added to the frontend by providing a control *palette*. A palette is a menu of buttons that can be defined by the user and that is displayed by a separate window in the frontend (see Figure 1). When one of the buttons is pressed, a user-defined command is executed in the kernel. In contrast to the "embedded control" solution, however, this command cannot carry any context information; the kernel itself therefore has to remember the current notebook state and update the notebook accordingly.

This solution does not require modifications in the notebook; instead a Mathematica program running in the kernel must remember and update notebook states accordingly. Still only simple control events without additional arguments can be handled in this way.

Remote Control. The interceptor process may be extended by a "remote control" with a corresponding interface that the lecturer uses for notebook control. This remote control generates messages that are transmitted to the kernel which subsequently triggers the corresponding updates in the frontend.

This solution shifts the functionality of the control palette from the frontend to the interceptor process. By an appropriate user interface, arbitrary user input may be incorporated into messages to the kernel. Moreover, the control interface itself can encapsulate part of the notebook state (e.g. the current position) and forward this information to the kernel (thus relieving the kernel program from this task).

The ultimate solution will probably consist of a combination of these possibilities; their appropriate mixture and area of application will be subject of investigation. In any case, some care has to be taken to prevent the individual session participants from modifying their notebook duplicates during the lecturer's presentation. For this purpose, the interceptor process may reset the state flag `Editable` when it receives a new notebook. On the other hand, the lecturer should not be allowed to perform operations via the standard frontend that are not appropriately propagated to the audience. There is no apparent technical solution to overcome this problem; as a pragmatic remedy we may display on the lecturer's host in a separate window the audience's view of the notebook such that the lecturer may detect inappropriate actions early.

5 Session Operation

Our basic model for the operation of a distributed education session is as follows:

1. The lecturer starts her local instance of the system consisting of frontend, kernel, and interceptor process.
2. Each other participant starts a corresponding local instance of the system (optionally without the kernel).
3. The interceptor of each participant gets connected to the interceptor of the lecturer.
4. The lecturer's interceptor subsequently *multicasts* messages to the interceptors of the audience.

In order to demonstrate the basic operation principles, we will start with a only-lecturer mode from which the distributed operation modes are derived.

5.1 Local Mode

When the education session is started, the lecturer triggers by her remote control the loading of a notebook from a file. The interceptor creates a corresponding message that lets the kernel execute a `NotebookOpen` command which in turn sends a message to the frontend to read in the corresponding notebook (see Figure 5).

Likewise, when the lecturer triggers a navigation event, the interceptor sends a message to the kernel to issue, e.g., a `SelectionMove` command. This in turn triggers a message from kernel to frontend to move the current notebook selection; this message is simply propagated by the interceptor.

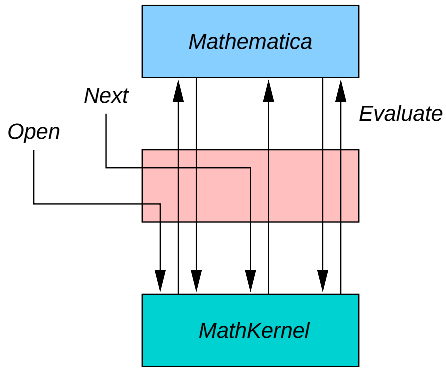

Figure 5: Local Mode

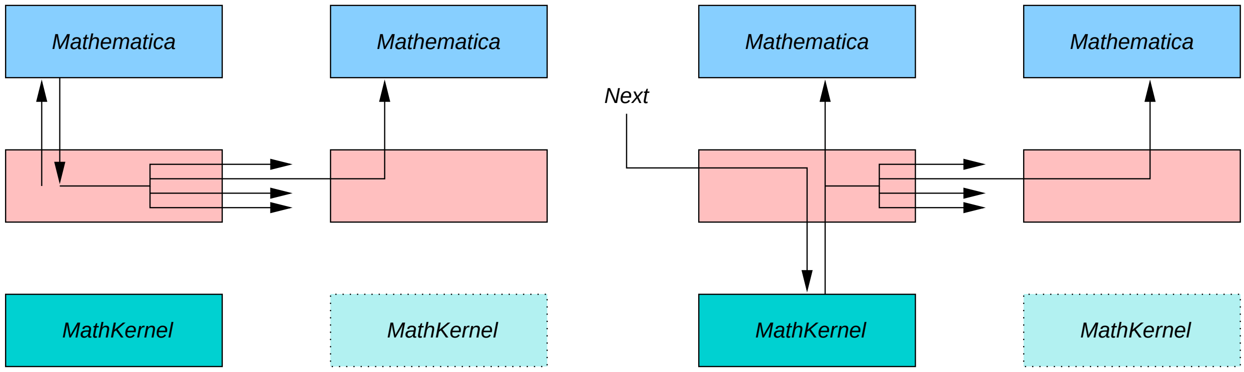

Figure 6: Controlled Mode

If the lecturer triggers the evaluation of some active notebook content in the frontend by pressing the `Shift-Return` key, the frontend generates a message which the interceptor forwards to the kernel; likewise the response message from kernel to frontend is propagated in the other direction.

5.2 Controlled Mode

In *controlled* mode, the lecturer presents a notebook to all participants connected to the lecturer's host. This is the default operation mode of a distributed education session.

When the lecturer triggers the loading of the notebook, initially the same operation sequence is performed as in local mode. However, after the frontend has read the notebook, the lecturer's interceptor simulates the kernel command `NotebookGet` by sending a corresponding message to the local frontend. The frontend responds by a message that contains the content of the notebook loaded. This message is multicast to the interceptors of all participants which in turn simulate the command `NotebookPut` that creates a notebook duplicate in the respective frontend (see left diagram in Figure 6).

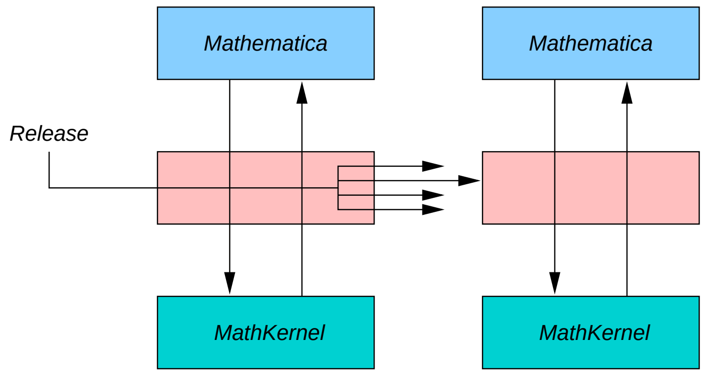

Figure 7: Independent Mode

In a similar way, every message triggered by the lecturer's kernel in response to a navigation command is not only propagated to the local frontend but also multicast to all other frontends. As a net effect, the lecturer controls the operation of *every* frontend. Consequently, in controlled mode the audience does not need local kernels for participation (see right diagram in Figure 6).

5.3 Independent Mode

In *independent* mode, the lecturer allows all participants to exercise on the notebook copies loaded in their local frontends.

When the lecturer switches from controlled mode to independent mode, a corresponding *release* message is multicast to the interceptors of all participants (see Figure 7). If no local kernel is available on some host, the corresponding participant remains in controlled mode.

Otherwise, the participant's interceptor issues a message corresponding to a `NotebookGet` command, receives the current notebook content from the frontend and saves it for later reuse. It then starts a local Mathematica kernel (if available and not yet started) and sends a message to the kernel to execute a `NotebookOpen` command. This command is responded by the interceptor (since the notebook is already loaded in the frontend) by a reply message corresponding to that one that was delivered by the lecturer's frontend after it had read in the notebook.

During independent mode, each participant's interceptor simply propagates messages from frontend to kernel and vice versa. The participant can thus experiment with active notebook contents, run examples, and update the notebook with corresponding results.

When the lecturer triggers return to controlled mode by a *hold* message, she may optionally also retrieve the updated notebooks stored in the frontends of the session participants. Each interceptor therefore sends to the local frontend a message simulating a `NotebookGet` command to which the frontend responds with the current notebook content. This content is returned

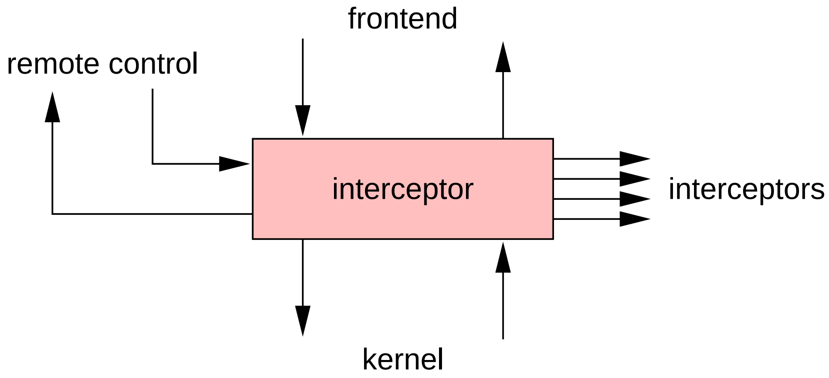

Figure 8: Interceptor

to the lecturer's interceptor and stored there in a file. Each interceptor returns to controlled operation by placing the original notebook (saved before switching into independent mode) into the frontend again.

5.4 Control Transfer

In controlled mode, the lecturer may also decide to temporarily transfer control to one of the participants (e.g. for presenting results).

The participant's system then may behave in a similar way as the lecturer's system controlling the operation of all other frontends. Different from the lecturer's behavior, however, the participant does not directly multicast messages to all other participants but forwards each message to the lecturer who in turn multicasts it. Consequently, only the lecturer's host needs to hold connection to the audience.

6 Interceptor Operation

The task of the *interceptor* process is threefold (see Figure 8):

- It must present itself to the frontend as a *kernel*.
- It must present itself to the kernel as a *frontend*.
- It must collaborate with one or many other interceptor processes and interact with its own control interface, kernel process and frontend process to implement a consistent distributed session.

The first two requirements ensure that the interceptor is *transparent* to both the frontend and the kernel. Thus the Mathematica system itself needs not be aware of the distributed application context; all distributed interaction is solely handled by the interceptor. The third requirement touches the actual core of the system's functionality.

The protocols that have to meet these demands were informally sketched in the previous sections. Formally, the interceptor process can be modeled as an input/output automaton [6] that accepts messages from any input channel (frontend, kernel, control interface, other interceptors) and acts accordingly by changing its internal state and/or sending output messages.

In the following, we will focus on one particular technical aspect of the interceptor process, the implementation of multicasting (one-to-many communication) by the interceptor on the lecturer's host to the hosts of the audience. This interceptor has to multicast the following types of data:

- Notebook contents (from frontend, large).
- Control messages and user input (from control interface and frontend, small to medium size).
- Computation results (from kernel, often small but potentially large).

The utilized multicasting scheme must be *reliable*, i.e., messages must not be lost, duplicated, or reordered, because otherwise inconsistencies among different notebook instances may occur. On the other hand, state changes of different notebook instances need *not* be synchronized, i.e., it is not necessary to wait until all participants have acknowledged the receipt of a message before the next message may be multicast. On the contrary, if the connection to one particular participant is slow or unreliable, the lecturer and the other participants should *not* be punished by an overall bad performance of the system.

While there exist a variety of fault-tolerant multicasting algorithms [9], a pragmatic solution that meets the demands sketched above is to apply multiple concurrent threads (see Figure 9). Each thread maintains a reliable connection to one particular host, independently reads the next multicast message from a queue and sends it to the respective receiver. Communication may be based on a reliable basic protocol with explicit message acknowledgments (TCP) or on an unreliable basic protocol (UDP) with application-level reliability established by, e.g., a sequence numbering scheme; each thread may also hold multiple communication lines with different protocols for the different kind of data sketched above.

By this form of multithreaded multicasting scheme, the various session participants are totally decoupled from each other. Moreover, additional participants may join the session late and

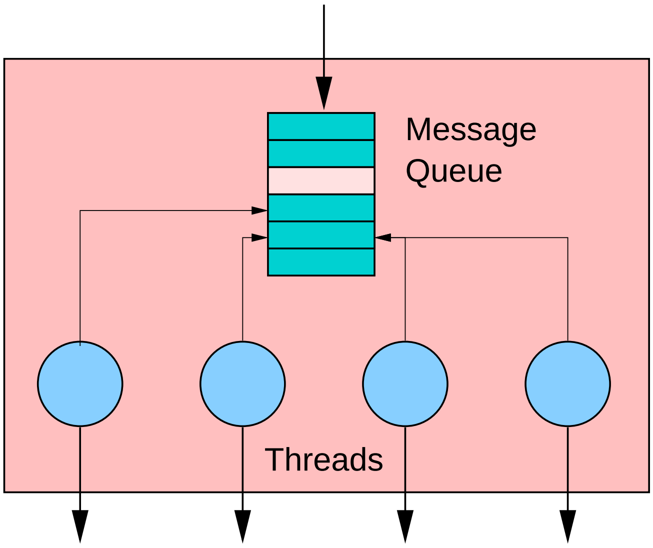

Figure 9: Multithreaded Multicasting

still experience the same sequence of events (represented by messages in the lecturer's queue). Of course, this raises the question when a message may be erased at all; two possibilities are:

- A message is erased from the buffer when all threads currently connected to the session have transmitted the message. A participant may join the session only at those points that put the local frontend in a state independent of the previous event history (e.g., when a new notebook is loaded).
- A message is buffered until all threads have transmitted a subsequent message that puts their respective frontends into a state independent of the previous history. A participant may join the session at any time since the history since the last such event can be replayed.

The methods sketched above may be implemented in any language or system with appropriate multithreading and networking facilities; Java is probably an excellent choice [5].

7 Conclusions

The presented design of a distributed education environment is constrained by the fact that the internal mechanisms of the Mathematica frontend cannot be easily accessed or modified; we have to cope with the (still extensive) facilities for external control provided by the MathLink interface. Nevertheless, by providing a rather limited system extension (a process that intercepts and multicasts communication between frontend and kernel), it is possible to support many features that are interesting for distributed education sessions. Augmented by public-domain video/audio multicasting support (MBone) it should be already possible to implement realistic distributed classroom scenarios. A prototype of the envisioned environment is currently being implemented in the frame of a diploma thesis. When it will be available, we will experiment with these facilities and start to develop appropriate education contents.

References

[1] *ATM Summer School*, Johannes Kepler University, Linz, Austria, July 1996. http://www.edvz.uni-linz.ac.at/KN/projekte/ss96/announce.html.

[2] Bruno Buchberger. Mathematica as a Rewrite Language. In Tetsuo Ida, editor, *Fuji Conference on Functional Logic Programming*, Shonan Village, Japan, November 1–4, 1996. Telos publishing conference.

[3] Bruno Buchberger et al. A Survey on the Theorema Project. In Wolfgang Küchlin, editor, *ISSAC'97, International Symposium on Symbolic and Algebraic Computation*, Hawaii, July 1997. ACM Press.

[4] Bruno Buchberger and Wolfgang Schreiner. CONCERT — A Software Architecture for Coordinating Education Sessions in Distributed Environments. Technical Report 97-04, RISC-Linz, Johannes Kepler University, Linz, Austria, February 1997.

[5] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Java Series. Addison Wesley, 1996.

[6] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, CA, 1996.

[7] Michael R. Macedonia and Donald P. Brutzmann. Mbone Provides Audio and Video across the Internet. *IEEE COMPUTER*, 27(4):30–36, April 1994.

[8] Kurt Maly, Hussein Abdel-Wahab, et al. Interactive Distance Learning over Intranets. *IEEE Internet Computing*, 1(1):60–71, January–February 1997.

[9] Sape Mullender, editor. *Distributed Systems*. Addison-Wesley, Wokingham, UK, 2nd edition, 1993.

[10] Distributed Electronic Classrooms – The MUNIN Project, 1996. http://munin.uio.no/English/intro.html.

[11] Stephen Wolfram. *The Mathematica Book*. Wolfram Media/Cambridge University Press, 3rd edition, 1996.