Process Centered Software Development on Mobile Hosts

— MS Thesis Proposal —

Peter D. Skopp
Department of Computer Science
Columbia University
500 West 120th St.
New York, N.Y.
10027

CUCS-035-93

October 11, 1993

Abstract

Software Development Environments have traditionally relied upon a central project database and file repository, accessible to a programmer's workstation via a high speed local area network connection. The introduction of powerful mobile computers has demonstrated the need for a new model, which allows for variable bandwidth machines as well as transient network connectivity to assist programmers in product development. A new client-server model is introduced which minimizes network traffic when bandwidth is limited. To support disconnected operation, I propose a process-based checkout model *by which process information and product files that may be needed during a planned period of dis-connectivity are pre-fetched with minimal user effort. Rather than selecting each file by hand, which is tedious and error-prone, the user only informs the environment of the portion of the software development process intended to be executed while disconnected. The environment is then responsible for pre-fetching the necessary files. It is hoped that these research efforts will enable programmers to continue working on a project without continuous high speed network access.*

1 Introduction

A multi-user software development environment (SDE) supports collaboration among multiple participants in large-scale software engineering projects. It provides a repository in which source code, object code, documentation, test cases, etc. reside, with some form of concurrency control to coordinate access to shared files. It integrates a collection of tools, ranging from editors and compilers to configuration managers and modification request systems, and generally tracks the progress of the project. A subclass of SDEs, called process-centered environments (PCEs), in addition provide some formalism through which a *process* may be specified — basically a partial ordering among software engineering tasks, constraints and obligations of those tasks, and the files and tools used in the tasks [15]. The generic PCE kernel is parameterized by the desired process written by the process architect, and the same PCE can support a wide range of different processes for different projects.

Marvel is a process centered software development environment in which the process is defined by a set of condition/activity/effects rules [9]. Marvel is constructed around an object oriented database (objectbase) which maintains persistent information about a process state. Marvel objects map to a file repository, and each object corresponds to a directory containing file attributes. An instance of Marvel represents its process internally by a rule network, whose links indicate possible forward and backward chains between rules related by a common predicate [8]. When a user requests to execute a particular software engineering task, Marvel employs the network to enforce and automate the subprocess involving the rule corresponding to that task. If the rule's condition — a complex logical clause — is not already satisfied in the objectbase, backward chaining attempts to execute other rules, one of whose effects might satisfy the original rule's condition. Its activity, usually the invocation of an external tool, cannot be initiated until the condition is true. After an activity completes, one of the rule's effects — each a sequence of predicates — is asserted, and forward chaining triggers any other rules whose conditions have now been met. There are multiple disjoint effects to reflect the multiple possible results of a tool invocation (i.e., various success and failure cases).

Each participant in a process interfaces to the system through a separate client, which supplies the user interface. The Marvel client is also responsible for invocation of external tools. The clients are coordinated by a server that incorporates the process engine, objectbase and the shared file repository [3]. The standard client/server protocol is for the client to display the repository in graphical format, and the user selects from a task menu and provides the desired arguments. The client then transmits this information to the server for any needed backward chaining. To execute an activity, the server submits the tool invocation information back to the client, and then goes on to accept the next message from its input queue.

In order to integrate unmodified tools into a Marvel environment, a generic wrapper called a shell envelope is used. The shell envelope resembles a UNIX shell script, but it is capable of both receiving input parameters from a Marvel client and sending a return value back, indicating the success or failure of an envelope. The client executes a shell envelope whenever a software task requiring tool invocation is performed. After the shell envelope finishes, the client sends the return code to the server. Based upon the return value, the server chooses which effect to assert, which eventually carries out any consequent forward chaining.

Marvel and other similar PCEs work well in an office environment where workstations are connected via a high speed network. The networking capabilities of these machines are used to facilitate communication between a PCE client and a server. A single file repository, maintained by the server, also forces a client to access all project related files via the network. Due to these constraints, PCE designers have virtually ignored the possibility of work without a high speed network, whose very existence was an underlying assumption of the PCE design. **Laputa** is an extension to Marvel which considers the full spectrum of network connectivity, in an attempt to support PCE usage on mobile hosts.

2 Motivation

Software engineers are well-known for their long working hours, some of which can be conducted at home using dumb terminals and modems. This mode of operation is relatively easy for an SDE to support — *if* one does not mind giving up many of the advantages of modern workstations, notably the large graphics displays. In theory, a full-scale workstation could be installed at home, but conventional modem speeds make it infeasible to treat this workstation as just any other node on the network. Low bandwidth serial line protocols such as SLIP and PPP are inadequate for maintaining a sophisticated display or transferring large files for local tool manipulation. X11 based protocols such as XRemote [4] and LBX [5] will maintain higher X11 throughput via a low bandwidth connection established between a pair of modems, but may still be too slow for interactive usage. To date, the SDE community has nearly ignored the possibility of off-site access, and the best that can be expected is a TTY user interface simulating the capabilities of the standard (graphics-based) user interface accessible only to users communicating over a local area network.

The recent proliferation of mobile computers, and sales predictions for the near future, point towards a large demand for not only mobile machines, but also for applications to run on these new computers. One obvious difference between desktop machines and their mobile counterparts is the stability and availability of network access. Most desktop machines in offices and labs are connected to high speed local area networks. In comparison, mobile machines may spend extended periods of time without network access, or connected but at reduced communication speeds. Software applications for mobile computers will have to dynamically adjust to changes in network latency, throughput, and general availability.

The advent of mobile computing thus introduces a new challenge for SDEs, and an exciting opportunity. Laptop or notebook computers provide essentially the same power as desktop workstations, but with low, perhaps varying bandwidth — and often operating in a completely *disconnected* state for arbitrary periods of time. The challenge is how to incorporate this emerging technology into multi-user environments that normally rely on (at least) a shared network file system. The opportunity is to completely rethink SDE architectures to consider the full spectrum of networking possibilities, multi-site as well as off-site, gigabit down to (temporarily) zero bandwidth, during normal operation.

3 Goals

The goal of **Laputa** is to investigate the problems of disconnected and low bandwidth PCE operation, and to develop a system which overcomes these problems. To facilitate low bandwidth operation, a new client/server model is introduced which re-thinks the responsibilities of the client in order to minimize network traffic. A user will be able to exploit wireless network connections by minimizing communication, hence reducing power consumption, and avoiding added connectivity expense. To support fully disconnected operation, a new checkout model is employed in which a user pre-fetches all project data and files needed to conduct planned work. The user is then able to remove a notebook computer executing a PCE client from the network for a period in order to conduct work off-site. It is assumed that the disconnected user restores the connection eventually, to merge the (partially) completed work with the ongoing efforts of other personnel collaborating on the same large-scale project. It is hoped that with proper planning and by using the more powerful **Laputa** client, a user will be able to take advantage of mobile computers, and continue to do productive work on a project in any location, regardless of network connectivity.

4 Related Work

Disconnected operation could have been achieved in the Sun Network Software Environment [1]. A user would select a software component to check out, and all of its constituent files were "acquired". The user was then able to work independently on the files in the component. Other users were free to "acquire" the same software component, increasing parallelism. On request or at "reconciliation" time, the system detected any changes in the file repository from the user's workspace, and copied the new versions of the checked out files. A `diff`-like tool assisted the user in merging the updated files with their newer versions. Unfortunately, it is often difficult for a user to correctly identify which files should be placed in each software component, and which components are actually needed. Furthermore, while pre-fetching large software components may enable a user to work in disconnected mode, the extremely weak concurrency control of Sun's NSE often leads to reintegration problems when more than one user has edited a file.

Numerous other SDEs employ some form of checkout model for concurrency control [2], but I know of none that either exploits the software process to assist in the selection of files to be checked out, or that permits disconnected operation.

Considerable research in disconnected file systems achieved through file pre-fetching has been done, but systems such as Ficus [6], Coda [12] and Tait's "File System for Mobile Computing" [16] were unable to draw upon the detailed application semantics inherently available from PCEs such as Marvel. Files were selected for pre-fetching either by explicit user selection, or by analyzing prior file access patterns. SunSoft is currently developing support for future versions of the Solaris operating system to allow for mobile hosts to operate in "disconnected" mode, but no detailed information about this research is available at this time. Since Sun's disconnected operation will be achieved at the file system level, it is a reasonable assumption that pre-fetch selection will be done in a similar fashion to the other disconnected file systems, and is unlikely to support process- based pre-fetching.

5 Low Bandwidth Operation

In order to minimize network bandwidth requirements, some of the traditional responsibilities of the client are re-targeted. To reduce the communication between a Marvel client and a server running on a remote host, it is important to understand the traffic produced by the current client/server architecture. The following is a list of the types of traffic generated:

- Objectbase information sent from the server to a client such as the current attribute values for an object.
- General system statistics sent from the server to a client such as who is currently using the system.
- Software tool invocation requests sent from the server to a client.
- Software tool responses sent from a client to the server.

- Client objectbase queries and rule firing requests sent from a client to the server.
- NFS traffic produced by a client executing a tool which requires accessing a file stored in the server's repository.[1]

Some preliminary work has already been done to reduce the network traffic sent from the server to a client. Specifically, the update information that the server sends to a client regarding changes made to the objectbase is minimized. Rather than sending a complete image of a Marvel objectbase to a client every time a change is made to an object, the server sends the client a `diff` with respect to the previous version of the objectbase. There is still a great deal of room for improvement in the other areas of traffic "production".

The greatest amount of network bandwidth consumed in the current Marvel client is due to the NFS traffic produced when a client executes a tool with a file which resides on a remote host. Even with the most sophisticated modems and compression techniques, transferring a large software source file over a telephone line for editing can take prohibitively long. A matched pair of 14,400 baud modems can only transfer about 1Kb of data per second[2]. Attempting to do more complex tasks such as compiling a file are virtually impossible due to the large number of input and output files (header files, object code, executables) that are processed.

5.1 Activity Management

The component of the Marvel client responsible for tool invocation is called the Activity Manager. The client passes activity invocation information to the Activity Manager, which then `forks` and executes a shell envelope which invokes the appropriate tool. Once a tool has finished executing, the Activity Manager passes return values back to the Marvel client, which in turn sends this information back to the server via the network.

In the context of a multi-user SDE operating over a high speed local area network, the

[1] Marvel is built upon NFS, so files from the server's repository may actually be NFS served by a host other than the machine running the Marvel server. In any case, we assume that the NFS host serving the Marvel clients and server is a non-mobile machine.

[2] Actual throughput varies depending upon the size of the file, which error correction protocol is used, and the quality of the phone line.

placement of the Activity Manager into the Marvel client was appropriate. By not burdening the server with activity invocation, as well as the added load a machine incurs from performing an activity, the server was able to operate more rapidly [3]. By distributing its computing load to client hosts, the overall power of the system was increased.

5.2 Proxy Client

When attempting to operate a Marvel client over a low speed network connection, the current location of the Activity Manager is a great hindrance to performance. In **Laputa**, the Activity Manger is removed from the low bandwidth client. Rather than having the Marvel server send an activity invocation message to a client, which would then produce NFS traffic to supply needed files to the client, the Activity Manager is relocated to run on a host local to the server. Local in this context implies a high bandwidth network connection to a host machine executing the server, not necessarily the same host. A new client called a Proxy is introduced, which contains the Activity Manager, but does not act as a user interface to the system.

In keeping with the previous Marvel design principles of load distribution, a Proxy may be started on any host, and each proxy client services only one user. A Proxy client registers with a server by sending a `PROXY_REGISTER` message to a running Marvel server. Once a proxy has been registered with the server, any tool invocation messages that would otherwise have been sent to a remote client are instead sent to the Proxy. When a Proxy receives an activity request message, it `forks` a child process and then executes the shell envelope which runs the external tool. The `STDIN`, `STDOUT` and `STDERR` file descriptors are redirected so that tool input and output appear to be coming from tools running on the local host, rather than from the Proxy. Once the activity is complete, the return values are sent back to the server. Minimal changes are made to the Marvel server so that activity requests are sent to the Proxy rather than the remote client, and to process incoming messages from the Proxy as if they had come from a traditional client.

5.3 Tool I/O

To support non-interactive text based tools such as a compiler, the low bandwidth client model works very well. Such tools are invoked by the Proxy, and all output messages appear on the screen of the **Laputa** low bandwidth client. It is important to note however that some tools such as editors are interactive, and often have graphical user interfaces. To run these tools correctly, the Proxy automatically sets its `DISPLAY` variable to point to the remote host at which the user is located. Once this step is taken, any graphic based tool will run on the Proxy host, but display its output and read its input from the remote host. This remote usage of interactive tools creates previously non-existent network traffic.

The network traffic generated by a complex graphic program can be immense. Since there are non-graphic based substitutes for many software engineering tools, the process architect (the person charged with writing the process definition) should be able to specify alternative tools for low bandwidth activities. Aside from the shell envelopes written for normal operation, low bandwidth envelopes may be written. When a Proxy client attempts to execute an envelope, it first searches for a low bandwidth version of the envelope to execute, and then defaults to the traditional envelope if no low bandwidth envelope is found. An example of a low bandwidth substitute for a graphic tool might be using the dbx debugger rather than xdbx, which is GUI based.

5.4 Low Bandwidth Summary

Low bandwidth operation in Marvel will be achieved by splitting the client into two components capable of running on different machines. This same principle in splitting a Marvel client into two components could easily be applied to PCEs and SDEs other than Marvel. A summary of the network traffic that the **Laputa** low bandwidth model creates is shown in figure 1. In this figure, we envision a Marvel server running on computer `A`. The user in this case is located at computer `B`. A Proxy client has been started on computer `C`. It is assumed that computers `A` and `C` share a high speed network, but that computer `B` is connected to the network via a low bandwidth connection.

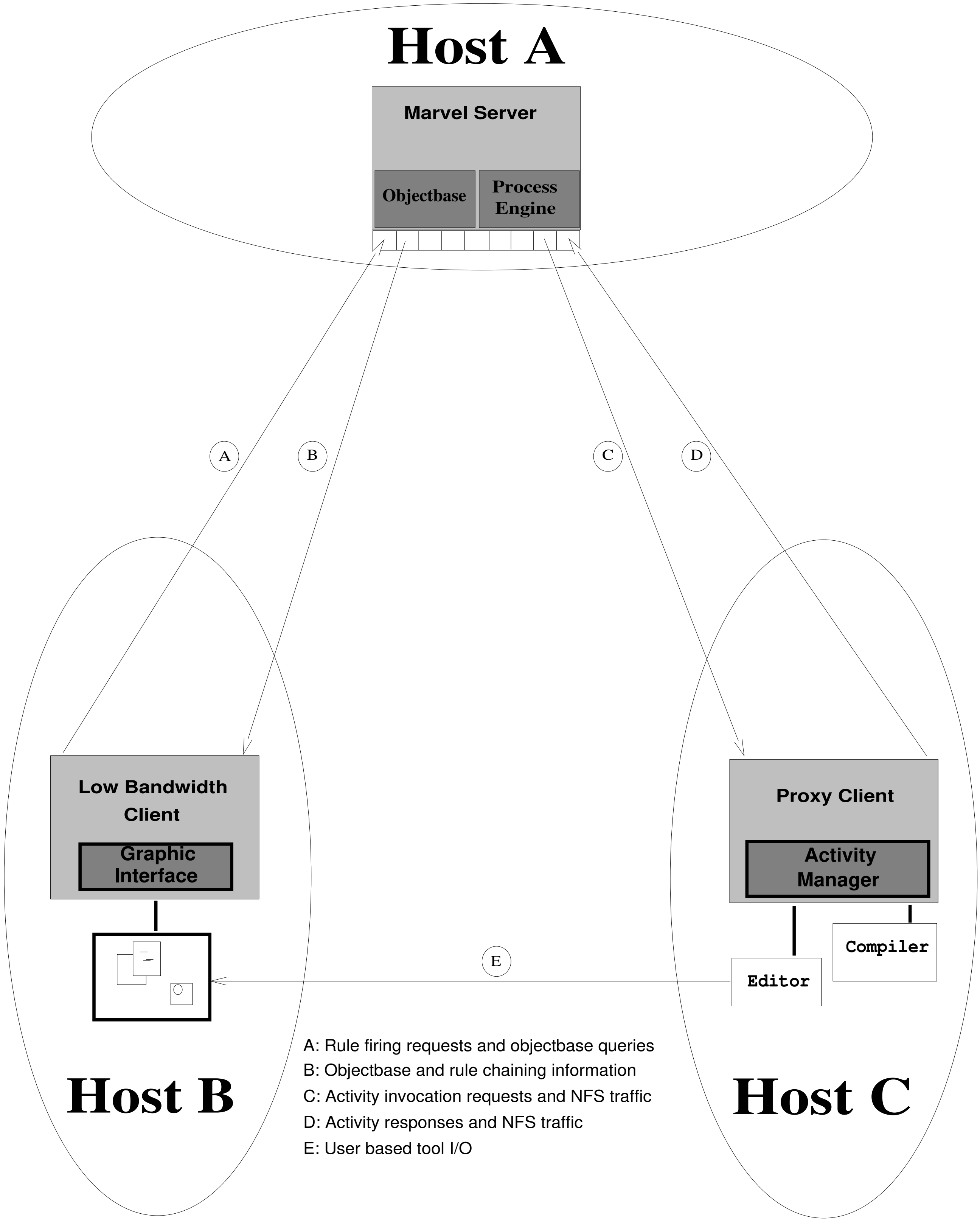

Figure 1: Summary of Low Bandwidth Client Network Traffic

There is another possible approach to low bandwidth operation. It is possible to run a Marvel client on a machine which shares a high speed network connection with a server, and which transmits the X11 based protocol generated by the user interface portion of the client to a remote machine. This approach shares the same problem of interactive tool I/O as the low bandwidth client proposed, because the software tools invoked by the client are still running on the remote machine, and must send all of their display information over the low bandwidth connection. Furthermore, even if the X11 protocol generated from the user interface were to be compressed via LBX [5], it would still use a greater amount of network bandwidth to operate than the proposed low bandwidth client needs.

6 Disconnected Operation

Even with an efficient low bandwidth client, we would often like to work on a project in a completely disconnected mode. This situation may arise when we do not have access to facilities that would allow us to establish a network connection from our current location. Even if it is always possible to establish a high speed network link (wireless computing), it may be desirable to work in a disconnected mode, for economic or security reasons. Although wireless computing promises to give mobile computers network access regardless of location, this service will not come for free. Many companies also "firewall" their networks as a security measure, making it impossible to connect to the network [13]. In these companies, disconnected operation is the only viable option for employees who are not on-site.

Since disconnected operation in **Laputa** is not viewed as a failure case, an alternative client will manage the explicit disconnection from the network and disconnected development. The **Laputa** disconnected client is a traditional client augmented with a single user process engine. Prior to disconnecting from the network, **Laputa** copies a complete copy of the objectbase from the server and stores a local copy of it on the mobile host. A subset of the files in the server's repository are also copied onto the mobile host. Once all of the data and file pre-fetching has completed, a user is able to remove a mobile computer from the network and use local copies of the objectbase and files. After a network connection is reestablished, files modified by a disconnected user are reintegrated into the server's repository.

6.1 Pre-fetching

Disconnected operation is achieved through a combination of intelligent data and file pre-fetching in conjunction with a single user Marvel process engine which replaces the functionality of a Marvel server. In order for pre-fetching to be effective in **Laputa**, I have formulated a list of requirements that the system must adhere to:

- Be able to pre-fetch a working subset of files such that the user may be able to do development work locally.
- The fact that files have been copied to a disconnected **Laputa** client should not hinder the work of *other* users.
- Inconsistencies between local copies of files and those in the central repository must be easily trackable upon reconnection.

Data pre-fetching is straightforward. A complete objectbase image is copied from the server. A user may attempt to conduct work while disconnected which requires accessing files which have not been pre-fetched. A complete copy of the objectbase is required in order to determine which files if any are missing from the local repository. While it is possible to prune some attributes from the objectbase in order to only maintain critical information about each object, the space savings from such an action would be minimal, as objectbases are relatively small. Experience using Marvel to develop large software systems has shown that a typical objectbase consumes less that .5% of the total file system space consumed by a project, and grows proportionally to the project's overall size.

Unlike data pre-fetching, it is usually not possible to pre-fetch an entire file repository onto a mobile machine due to space limitations. I propose three possible approaches to the selection of files to pre-fetch, the last of which is a novel contribution of this research:

- *Manual*: A user supplies an explicit list of objects whose files should be pre-fetched.

- *Heuristic*: The system maintains statistics about each user's past efforts, and assumes the same files are needed for future work. Statistics which might be kept include each individual user's recent file accesses, as well as which tasks each user has recently performed.

- *Process-based*: A user supplies an explicit list of tasks to be carried out while disconnected, and the system analyzes the process definition to determine which files are required for those tasks, as well as all tasks which the system may initiate due to chaining, as described earlier in section I.

In **Laputa**, I will implement both *manual* and *process-based* selection, but not *heuristic* selection. Both *manual* and *heuristic* selection have major problems, which limit their effectiveness, however *manual* selection will be implemented as a developmental stepping stone towards the more complex *process-based* selection.

An entirely *manual* approach puts the full burden of selection on the user. If a critical item is discovered to be missing after a the network connection has been broken, local development may be stalled. While I assume a user will be able to identify at a high level what type of work is desired to be accomplished during a planned period of dis-connectivity, a user may not always correctly identify all needed support files. An example would be a software engineer who pre-fetched some "C" source object's file to edit, but neglected to pre-fetch all of the header files required to recompile the source files. In this case, recompilation of the pre-fetched "C" file would be impossible, and the development would be limited to editing the file, but not recompilation. In a large project, it would be easy for some needed files to be forgotten.

Pure *heuristic* selection assumes inertia on the part of the user. That is to say, the system generally prepares for a user to continue doing essentially the same work while disconnected that was recently being performed while connected. Each user's recent development efforts would be tracked. It is assumed that any file which was accessed during recent development, would also be needed for future efforts. The system would keep track of which files were most recently accessed by each user. There is a tradeoff here between biasing the heuristics towards pre-fetching too little versus too much. If the user does not plan to repeat exactly the same task (which may have already been finished), the materials available may not be sufficient for the new work. Yet on a portable machine with limited disk space, we would like to prune out all unnecessary files from the local disk in order to preserve the precious commodity.

Process-based selection addresses the problems encountered with *manual* and *heuristic* selection. However, the process-based approach is not as simple as it sounds. Practical industrial-scale processes are complex, with numerous opportunities for choice or iteration [10]. Since each rule may have conditions which lead to backward chaining and effects which lead to forward chaining, we can construct a graph of a portion of the process, showing the relationships amongst specific rules. Expanding the graph to show the transitive closure of consequences emanating from a single rule can lead to very large graphs, and instantiating each rule's task with the appropriate files could mark most of the repository for pre-fetching. Figure 2 shows a very simple and small transitive closure of a possible rule network.

We combine process knowledge with heuristics from previous access patterns in order to prune the branching paths, producing the subset of files most likely to be needed. **Laputa** will always pre-fetch files required to fulfill any tasks which are reached through backward chaining, before those needed to support forward chaining of a user identified task. The ordering is meant to assure that the system has pre-fetched all files required to perform the task which was originally identified by the user prior to disconnection. If disk space is limited, the system makes sure that all files required to fulfill backward chaining from a desired task are pre-fetched, at the possible expense of files which would be required for tasks reached through forward chaining. In this way, we ensure that the user is able to at least perform the tasks which were identified by the user prior to disconnection.

In **Laputa**, we will maintain statistics on which of the multiple effects of a rule has been selected most frequently, with respect to this specific user and the arguments desired for the originating task. If statistics are available for a specific task, the information is used to restrict the expected forward chaining to a manageable level. By weighting the links connecting tasks in the rule network based on the statistical probability that a path will be followed, we can compute a probability graph. Once the rule graph is generated and the

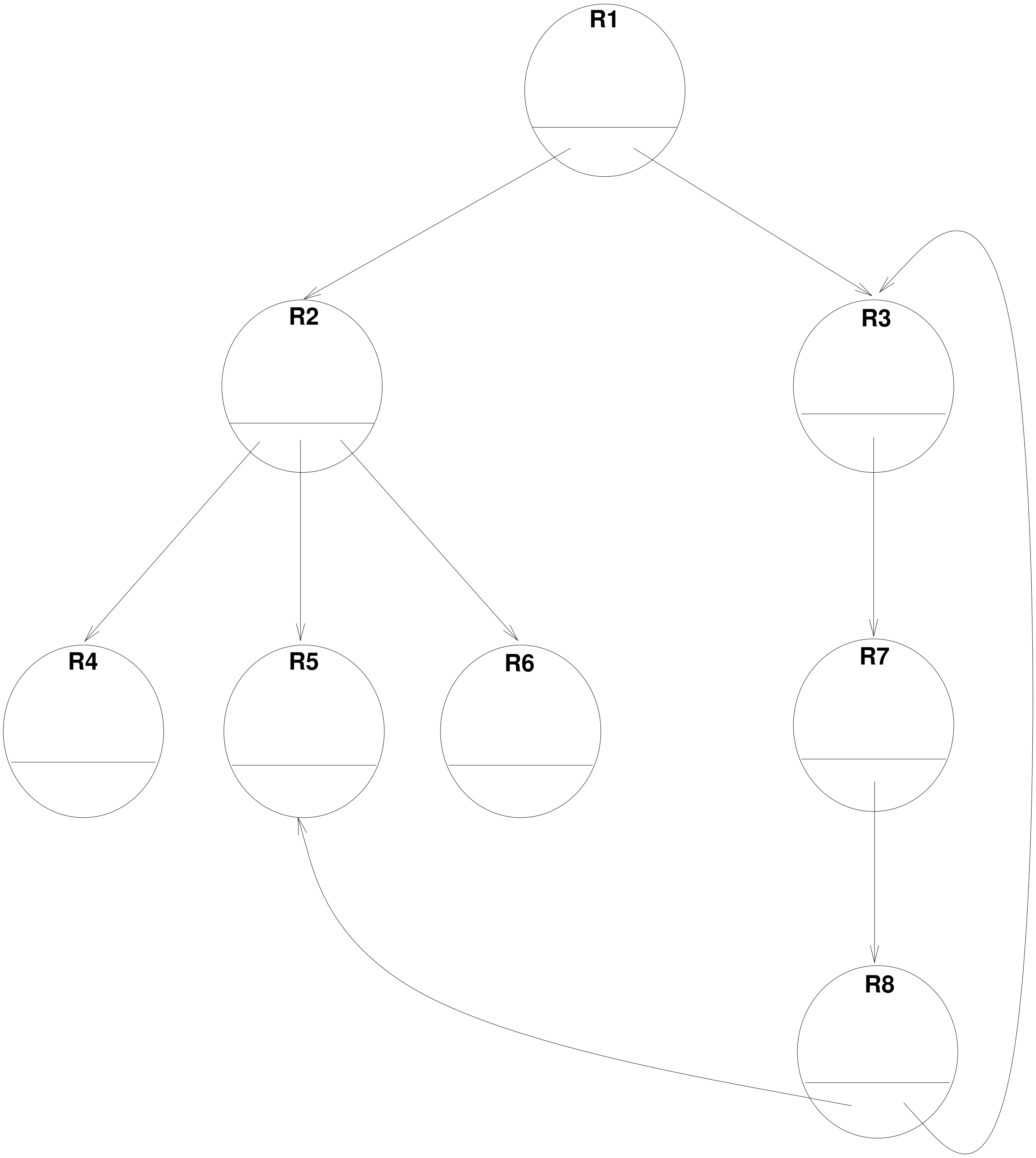

Figure 2: Transitive Closure of Rule Set

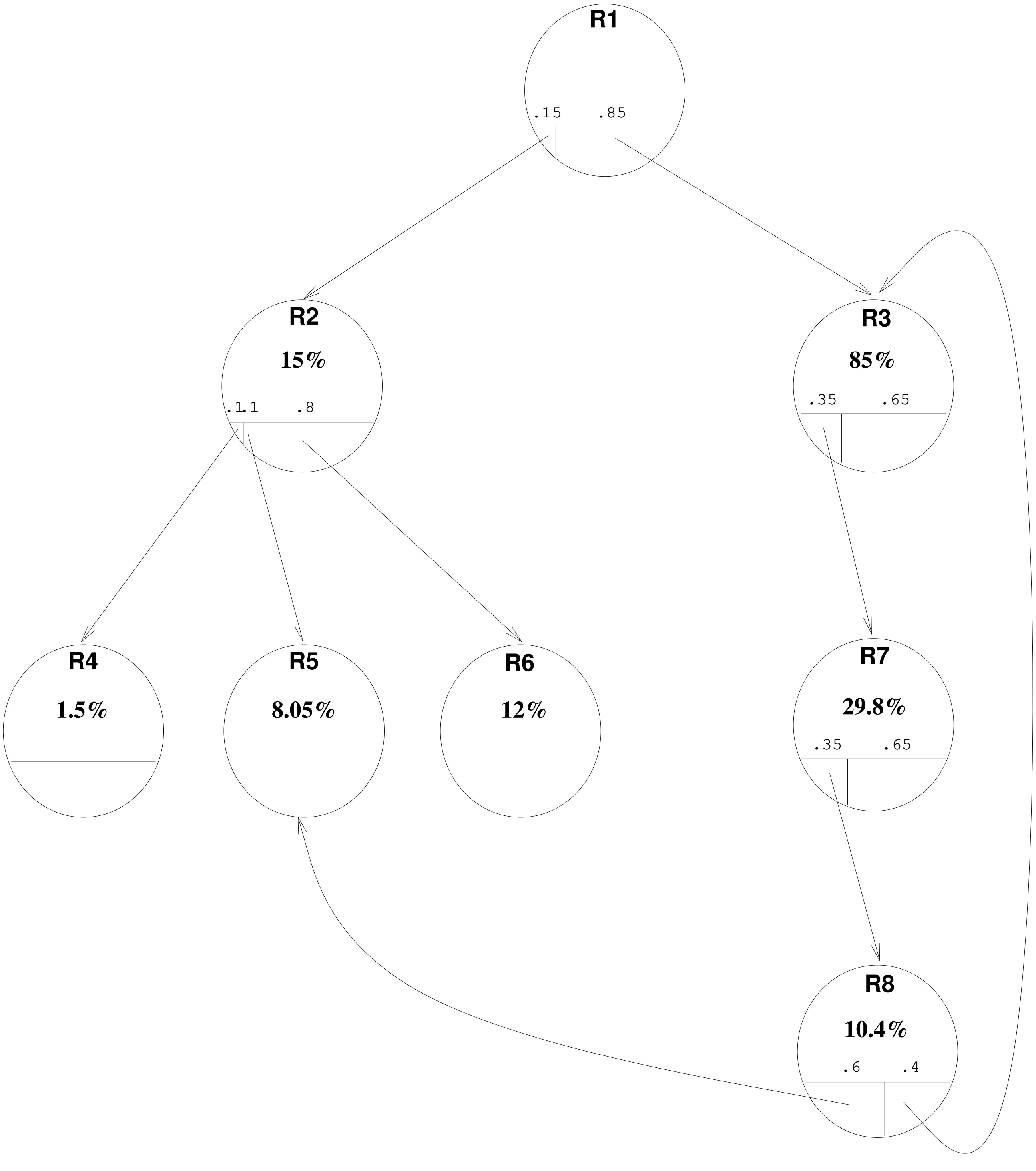

Figure 3: Probability Graph

probability of transition-ing from each node of the graph is identified, we can compute the likelihood of reaching each node in the graph. The probability graph of a sample rule network is shown in figure 3. If the system guesses incorrectly, and a rule which was identified as unlikely to be reached via forward chaining is in fact fired, the forward chaining must be delayed until reconnection when the needed files are accessible from the server's repository. The degree to which file selection is pruned can also be adjusted to accommodate the varying size of a local disk, e.g., by not completing even the most likely forward chaining path if the disk is too small and considering multiple paths if there is more free space.

Laputa allows for a process architect to distinguish between `long duration` and `short duration` tasks. An example of a `long duration` task might be editing a file, where as a `short duration` task might be removing a file from a revision control system. The distinction between `long duration` and `short duration` tasks further helps **Laputa** prune which files get pre-fetched. Upon a request to disconnect, the **Laputa** process engine follows the backward chains emanating from a desired rule firing, and attempts to execute any needed tasks to satisfy the original rule. If a rule's task is of type `short duration`, then the actual time consumed by executing this task will be small, and it is done prior to disconnection. This differentiation is important as it allows us to do necessary work prior to disconnection, hence reducing the number of files that need to be pre-fetched in order to satisfy backward chaining. The process engine stops firing rules as soon as a `long duration` task (such as editing) is encountered.

6.2 Concurrency Control

Once pre-fetching is complete and the network link is broken, we encounter the concurrency problems associated with having multiple copies of a file. The obvious approach to concurrency control in this context would be the "checkout" model found in most version control tools and some modern database systems (e.g., [14, 11]). In this model, each pre-fetched file would be locked in either a `read only` or `writable` mode, depending on whether the file is only needed for reading or possibly may be updated during the disconnected process fragment. These locks would be maintained persistently until later reconnection and “checkin”.

But a more flexible approach is desirable for some software engineering applications [2]. Fortunately, in addition to being parameterized by the desired process, Marvel includes a sophisticated approach to concurrency control whereby new lock modes, compatibility among lock modes, and resolution of locking conflicts can also be defined on a project-specific basis [3, 7]. Locks in Marvel are applied to an object in the objectbase, rather than directly on the files associated with an object. These facilities are exploited to support the **Laputa** disconnected client.

Laputa introduces a new set of persistent locks to Marvel's objectbase. Objects which contain read-only files are locked in a new `dirty read` mode and the files are replicated on the **Laputa** client; unlike `shared` mode, `dirty read` is defined to be compatible with the `exclusive` mode so that other users can continue to work on the file. An obvious example of objects and their associated files that could be locked in `dirty read` mode are “C” header files that the user does not intend to edit, but which are needed to compile a modified “C” source object's file. The use of the `dirty read` lock would allow other users to make modifications, and the disconnected user would continue to use the outdated version of the file until reintegration occurred.

Objects which contain write-able files can be locked in one of two modes, `creative exclusive` or `generated exclusive`. **Laputa** allows a process architect to describe a task as either `creative` or `generated`. A `creative` task is one that involves an interactive tool, usually one that produces a valuable product, such as an editor or a drawing program. These tasks are separated from `generated` tasks whose output can easily be reproduced without direct user involvement. `Generated tasks` will typically read in one or more input files, process the input, and produce one or more output files without modifying the inputs. Examples of `generated` tasks are assembling, compiling, and linking because the tools used in these tasks can easily be invoked to re-create their output. If a file is pre-fetched in order to be modified by a `creative` task, the file is locked in `creative exclusive` mode. Otherwise the file is locked in `generated exclusive` mode.

The two `exclusive` modes are useful during reintegration. Due to the `dirty read` mode

	CX	GX	DR	S
CX	**no**	**no**	**yes**	**no**
GX	**no**	**no**	**yes**	**no**
DR	**yes**	**yes**	**yes**	**yes**
S	**no**	**no**	**yes**	**yes**

CX = Creative Exclusive
GX = Generated Exclusive
DR = Dirty Read
S = Shared

Figure 4: Laputa lock matrix

used in **Laputa**, it is possible for a pre-fetched file to become inconsistent with the version stored in the server's repository. Furthermore, files which are locked in `generated exclusive` mode may have been generated with one of the inconsistent files. Files from objects locked in `creative exclusive` mode are always "dominant", i.e., they will always be considered the most recent copy of a file and so can always safely replace older versions in the shared file repository upon reintegration. Because `generated` tasks invoke tools that read input files from objects locked in `dirty read` mode, some caution must be exercised when re-integrating these files, to assure that all `generated` files are consistent with their respective input files as found in the repository. The lock matrix shown in figure 4 summarizes the compatibility between the various lock modes introduced by **Laputa**.

6.3 Reintegration

A network connection can be re-established by the disconnected user at any time desired, at which point reintegration begins. Reintegration first detects any changes between the objectbase attributes of objects locked in `dirty read` or `generated exclusive` mode. If no differences are found, the files locked in both `creative exclusive` and `generated exclusive` modes can be presumed valid — and are copied into the repository, overwriting the previous versions.

But if some shared object attributes have changed, then reintegration occurs in four stages:

1. All files from objects locked in `creative exclusive` mode are copied into the shared repository, replacing previous versions.

2. All files from objects locked in `generated exclusive` mode that are dependent[3] upon files from objects locked in `dirty read` mode, which in turn are different from the versions stored in the shared repository, are deleted.

3. All files of objects locked in `generated exclusive` mode, which are dependent upon objects locked in `generated exclusive` mode but which contain files that were deleted in step 2, are deleted. This step iterates through the complete transitive closure.

4. All remaining objects locked in `generated exclusive` mode are presumed to be valid and their files are copied into the shared repository.

Once all of the files updated in the **Laputa** client have been copied into the main repository, all files which were part of objects locked in `generated exclusive` mode and deleted in steps 2 and 3 are regenerated by the process. This step is possible because only files generated by tools (without direct user intervention) have been deleted. The process engine is thus capable of triggering the appropriate tasks to regenerate all of the missing files, finishing reintegration.

6.4 Dependency Tracking

The reintegration phase of the **Laputa** disconnected client relies on the ability to detect dependencies between objects. Unfortunately, this information is not readily available from only the process description. To test for object dependency, a virtual copy of the server's objectbase is created in the server's memory. Objects that contain modified files are integrated into the virtual objectbase one at a time. After each modified object has been integrated into the virtual objectbase, forward chains are followed to bring the virtual objectbase into a consistent state. At each stage, the virtual objectbase is compared against the original objectbase maintained in the server. When an attribute of an object changes following a different object's virtual reintegration, we note the change and presume a dependency between

[3] A dependency exists between two files, when one file is read in as input to a tool that either produces the second file, or produces a file upon which the second file is dependent.

the two objects. The virtual reintegration continues until all object dependencies have been found.

7 Contributions

The contributions of this thesis will be the following:

1. A new low bandwidth PCE model.

   The low bandwidth client and proxy client introduced attempt to reduce the network bandwidth consumed by a Marvel client in order to make operation over low speed or high cost network connections more feasible. The implementation will be specific to Marvel, although the model of operation should be applicable to many other SDEs.

2. Disconnected operation achieved through process-based file pre-fetching, and process-based reintegration.

   Previously developed systems which attempted to facilitate disconnected operation through file pre-fetching were only able to use very weak heuristics to determine which files to pre-fetch. **Laputa** introduces the notion of process-based pre-fetching. Using process knowledge, the system is capable of identifying all files required to perform a desired software engineering task while in disconnected mode. The system is also able to reconcile inconsistencies between different versions of a file, by using the process definition. This pre-fetching model should be applicable to a wide range of PCEs.

8 Schedule

The following is a rough schedule of milestones to be reached in this research.

November-93	Completion of low bandwidth and proxy clients.
March-94	Single user process engine operational in disconnected mode.
April-94	Marvel server augmented with persistent locks, and maintains

	statistics about past development efforts.
June-94	Process based file selection implemented.
August-94	Process based reintegration implemented.
October-94	Write and defend dissertation.

9 Status

The **Laputa** implementation is currently in progress. The initial platform for the notebook computer will be a SparcBook 2 with 500MB disk, running SunOS 4.1.2. Marvel 3.1 itself consists of about 150,000 lines of C, lex and yacc, and runs on SparcStations, DECStations and IBM RS6000s. It was released in March 1993, and has been licensed to approximately twenty institutions to date.

10 Acknowledgements

I would like to thank Dan Duchamp, who initially interested me in the problems of mobile computing and Gail Kaiser, who eagerly embraced and motivated this research.

References

[1] Evan W. Adams, Masahiro Honda, and Terrence C. Miller. Object Management in a CASE Environment. In *11th International Conference on Software Engineering*, pages 154–163, Pittsburgh PA, May 1989. IEEE Computer Society Press.

[2] Naser S. Barghouti and Gail E. Kaiser. Concurrency Control in Advanced Database Applications. *ACM Computing Surveys*, 23(3):269–317, September 1991.

[3] Israel Z. Ben-Shaul, Gail E. Kaiser, and George T. Heineman. An Architecture for Multi-User Software Development Environments. *Computing Systems, The Journal of the USENIX Association*, 6(2):65–103, Spring 1993.

[4] David Cornelius. XRemote: A Serial Line Protocol for X. In *6th Annual X Technical Conference*, January 1992.

[5] Jim Fulton and Chris Kent Kantarjiev. An Update on Low Bandwidth X (LBX), A Standard for X and Serial Lines. Technical Report P93-00001, Xerox Palo Alto Research Center, February 1993.

[6] J. S. Heideman, T. T. Page, R. G. Guy, and G. J. Popek. Primarily Disconnected Operation: Experiences with Ficus. In *Second Workshop on Management of Replicated Data*. IEEE, November 1992.

[7] George T. Heineman. A Transaction Manager Component for Cooperative Transaction Models. Technical Report CUCS-017-93, Columbia University Department of Computer Science, July 1993.

[8] George T. Heineman, Gail E. Kaiser, Naser S. Barghouti, and Israel Z. Ben-Shaul. Rule chaining in MARVEL: Dynamic binding of parameters. *IEEE Expert*, 7(6):26–32, December 1992.

[9] Gail E. Kaiser, Peter H. Feiler, and Steven S. Popovich. Intelligent Assistance for Software Development and Maintenance. *IEEE Software*, 5(3):40–49, May 1988.

[10] Gail E. Kaiser, Steven S. Popovich, and Israel Z. Ben-Shaul. A Bi-Level Language for Software Process Modeling. In *15th International Conference on Software Engineering*, pages 132–143, Baltimore MD, May 1993. IEEE Computer Society Press.

[11] Won Kim, Nat Ballou, Jorge F. Garz, and Darrell Woelk. A Distributed Object-Oriented Database System Supporting Shared and Private Databases. *ACM Transactions on Information Systems*, 9(1):31–51, January 1991.

[12] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *Proc. Thirteenth ACM Symp. on Operating System Principles*, pages 213–225. ACM, October 1991.

[13] Marcus J. Ranum. A network firewall. In *World Conference on Systems Management and Security*, 1992.

[14] M. J. Rochkind. The Source Code Control System. *IEEE Transactions on Software Engineering*, SE-1:364–370, 1975.

[15] *2nd International Conference on the Software Process: Continuous Software Process Improvement*, Berlin, Germany, February 1993. IEEE Computer Society Press.

[16] Carl D. Tait and Dan Duchamp. Detection and Exploitation of File Working Sets. In *11th International Conference on Distributed Computing Systems*, pages 2–9. IEEE, May 1991.