Disconnected Operation in a Multi-User Software Development Environment

Peter D. Skopp *
Gail E. Kaiser †

Columbia University
Department of Computer Science
New York, NY 10027

CUCS-026-93
August 1993

Abstract

Software Development Environments have traditionally relied upon a central project database and file repository, accessible to a programmer's workstation via a local area network connection. The introduction of powerful mobile computers has demonstrated the need for a new model, which allows for machines with transient network connectivity to assist programmers in product development. We propose a *process-based* checkout model by which process and product files that may be needed during a planned period of disconnectivity 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. We hope that this approach will enable programmers to continue working on a project without network access.

To appear in the *IEEE Workshop on Advances in Parallel and Distributed Systems*, Princeton NJ, October 1993.

©1993 Peter D. Skopp and Gail E. Kaiser

Keywords: Disconnected Operation, software development pre-fetching, mobile computing.

*Skopp is supported in part by the National Science Foundation.
†Kaiser is supported by grants from the National Science Foundation, Andersen Consulting, Bull HN Information Systems and IBM Canada Ltd, and by the New York State Center for Advanced Technology in Computers and Information Systems.

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 [1]. The generic PCE kernel is parameterized by the desired process, and the same PCE can support a wide range of different processes.

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 [2] and LBX [3] will maintain higher throughput via a serial line, 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 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 zero bandwidth, during normal operation.

2 Laputa Overview

In the **Laputa** project, we are primarily investigating the problems of disconnected operation, when a software engineer removes a notebook computer from the network for a period in order to conduct work off-site. (In the related **Oz** project, we are studying geographically distributed operation over a high-speed network connecting multiple sites [4].) We assume the 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. We have chosen the PCE subclass of SDEs, where we can exploit a predefined process to partially automate the selection of files to download prior to disconnection and to structure the off-line work as a well-understood fragment of the overall process.

Laputa is being implemented by modifying the Marvel 3.1 environment, in which the process is defined by a set of condition/activity/effects rules [5]. 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 [6]. 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, backward chaining attempts to execute other rules, one of whose effects might satisfy the original rule's condition. Its activity, usually 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 and forks individual tools. The clients are coordinated by a server that incorporates the process engine and the shared file repository [7]. The standard client/server protocol is for the client to display the repository in graphical format, the user selects from the task menu and clicks on the desired arguments, and then the client 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 goes on to accept the next message from its input queue. After terminating the tool execution, the client returns the results to the server, which eventually carries out any consequent

forward chaining.

In the **Laputa** extension of Marvel, we are developing an expanded client that takes over the behavior of the server during the time when the computer executing that client is disconnected from the network. This client maintains a local, single-user process engine and file manager that duplicate portions of the rule network and repository from the main server. These are populated in anticipation of explicit disconnection, and any changed files will be reintegrated during later reconnection.

3 Pre-Fetching

Disconnected operation is achieved through intelligent file pre-fetching. In order for pre-fetching to be effective in **Laputa**, we have formulated a list of requirements that the system must adhere to:

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

Disconnected operation supported through file pre-fetching is not a new area of research, however previous systems [8, 9, 10] were unable to draw upon the detailed application semantics inherently available from PCEs such as Marvel. We see three possible approaches to the choice of files to pre-fetch, the last of which is a novel contribution of this research:

1. *Manual*: A user supplies an explicit list of files to be pre-fetched.
2. *Heuristic*: The system maintains statistics about each user's past efforts, and assumes the same files are needed for future work.
3. *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 the files required for those tasks, their prerequisites and their consequences.

All three methods of selection will be implemented in **Laputa**, although *manual* and *heuristic* selection have major limitations.

An entirely *manual* approach puts the full burden of file selection on the user. If a critical item is discovered to be missing after a portable computer has broken its network connection, the local development effort may come to a halt. While we 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 support files. An example would be a software engineer who pre-fetched some "C" source files to edit, but neglected to pre-fetch all of the header files required to recompile the source files. In a large project, it would be easy for some needed files to be forgotten, hence stalling development.

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. 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 [11]. The transitive closure of consequences emanating from a process step can be immense, and instantiating each enclosed task with the appropriate files could mark most of the repository for pre-fetching. Combining process knowledge with heuristics from previous access patterns is thus useful to prune the branching paths, producing the subset of files most likely to be needed. We always pre-fetch files required to fulfill any constraints for a given task before those required for the obligations following that task. The ordering is meant to assure that a user has all files required to perform a desired task, at the possible expense of tasks to be initiated after the original task had completed.

For example, in the Marvel context it seems appropriate to maintain statistics on which of the multiple effects of a rule has been selected most frequently, with respect to this specific user and/or the arguments desired for the originating task, to restrict the expected forward chaining to a manageable level. When the system guesses incorrectly, the forward chaining must be delayed until reconnection. 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.

4 Concurrency Control

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., [12, 13]). Each pre-fetched file would be locked in `shared` or `exclusive` mode, depending on whether it is only to be read 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 [14]. 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 [7, 15]. We exploit these facilities to support the **Laputa** disconnected client.

Read-only files can be locked in a new `dirty read` mode and 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 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 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.

Write-able files can be locked in one of two modes, `creative exclusive` or `generated exclusive`. The **Laputa** extension of Marvel allows a process architect (the person charged with writing the process definition) to describe a task as either `creative` or `generated`. A `creative` task is one that involves an interactive tool that produces a valuable product, such as an editor or a drawing program. These tasks are differentiated from `generated` tasks whose output can easily be reproduced without direct user input. `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. When a file is pre-fetched in a write-able mode, if the task that requires the file is `creative`, then the file is locked in the `creative exclusive` mode. Otherwise the file is locked in `generated exclusive` mode.

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

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

Figure 1: Laputa lock matrix

The two `exclusive` modes are useful during reintegration. Files 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 locked in `dirty read` mode, some caution must be exercised when reintegrating 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 1 summarizes the compatibility between the various lock modes relevant for pre-fetched files.

5 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 shared file repository and the local copies 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 files had indeed changed, then the reintegration occurs in four stages:

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

2. All files locked in `generated exclusive` mode that are dependent upon files locked in `dirty read` mode, which in turn are different from the versions stored in the shared repository, are deleted. We say that a dependency exists between two files, when one file is read in as input to a tool that produces the other file as output.

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

4. All remaining files locked in `generated exclusive` mode are presumed to be valid and 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 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 Example

The example in this section, although necessarily abbreviated, is intended to be typical of **Laputa** usage. We start by presenting a piece of a sample software development process. The process definition is written in a language similar to the one supported by Marvel, but the syntax here is intended to be more readable. We have also added a few features specifically for **Laputa**, as noted below.

The `compile` rule in figure 2 operates on "C" source files, called `cfiles`. The condition on firing the rule is that the file has not already been compiled, and that the file is "referenced". Referencing in this context refers to analyzing a `cfile` to determine the set of header files that it will include upon compilation. There are two possible effects of this rule, corresponding to either success or failure by the compiler.

The `edit` rule in figure 3 is used to edit "C" source files. The condition on firing this rule specifies that the file must be "reserved" and that the person who is trying to edit the file must be the same person who reserved the file. The binding section of the rule finds all executable files derived from the `cfile`. Aside from the obvious effects such as changing the file's timestamp, all executable files that were bound to the `exefiles` variable are marked as `Not_Built`. It

```
rule COMPILE: cfile

conditions:
  cfile.compile_status = ``Not_Compiled'';
  cfile.reference_status = ``Referenced'';

action: generated:
  return_value =  compile cfile;

effects:
  if return_value = Error
    cfile.compile_status = ``Error'';
  else cfile.compile_status = ``Compiled'';
```

Figure 2: Compile Rule

is important to note that this rule's action (activity) is `creative` as opposed to the other rules shown, which are `generated`.

The `reserve` rule in figure 4 is used to reserve a file from a version control system such as RCS [16]. (Note that the use of such a tool is orthogonal to the concurrency control system and lock compatibility matrix supported by the Marvel kernel, and its use in checking files in and out of a **Laputa** client. In particular, an entire RCS delta file could potentially be copied by **Laputa** during pre-fetching and reintegration, just like the input and output arguments of any other tool.) The condition to firing this rule is that the file being reserved is currently "available", and the effects change the reservation status and holder attributes of the file.

The `reference` rule in figure 5 finds all header files that a given `cfile` includes. The list of header files is then stored in the headers attribute for use by the compiler.

The `build` rule in figure 6 first checks to make sure that an executable file does not already exist, to avoid redundant work, and that all of the object code needed to build the executable is available. The `test` rule in figure 7 simply verifies that an executable files exists, and then runs it through a test sequence.

A software engineer following a process like the one described above may wish to disconnect from the network. This user would give **Laputa** a list of rules instantiated with their arguments, indicating the planned task, and then the system would determine the other rules that might need to be fired to satisfy the conditions as well as the other rules that might be triggered by the effects.

To satisfy the `edit` rule, the file must be reserved by the current user. If this condition is not already met,

```
rule EDIT: cfile

conditions:
  cfile.reservation_status = ``Reserved'';
  cfile.reservation_holder = Current_user;

action: creative:
  edit cfile;

bindings:  exefiles = all EXEFILE such that
             linkto[cfile, EXEFILE];

effects:
  cfile.compile_status = ``Not_Compiled'';
  cfile.time_stamp = Current_time;
  cfile.referenced = ``Unreferenced'';
  for all exefile in exefiles
    exefile.build_status = ``Not_Built'';
```

Figure 3: Edit Rule

```
rule RESERVE: cfile

conditions:
  cfile.reservation_status = ``Available'';

action: generated:
  checkout cfile;

effects:
  cfile.reservation_status = ``Reserved'';
  cfile.reservation_holder = Current_user;
```

Figure 4: Reserve Rule

```
rule REFERENCE: cfile

conditions:
  cfile.reference_status = `Unreferenced'';

action: generated:
  headers = find_dependencies cfile;

effects:
  cfile.headers = headers;
  cfile.reference_status = ``Referenced'';
```

Figure 5: Reference Rule

```
rule BUILD: exefile

bindings: cfiles = all CFILE such that
            linkto[exefile, CFILE]

conditions:
  exefile.build_status = ``Not_Built'';
  for all cfile in cfiles
    cfile.compile_status=``Compiled'';

action: generated:
 /* the object code file is treated as
    an attribute of the source file */
  return_value = link exefile cfiles.obj;

effects:
  if return_value = Error
    exefile.build_status = ``Error'';
  else exefile.build_status = ``Built'';
```

Figure 6: Build Rule

```
rule TEST: exefile

conditions:
  exefile.build_status=``Built'';

action: generated:
  test exefile;
```

Figure 7: Test Rule

the system attempts to find other rules whose execution may satisfy the initial rule. An inspection of the rule network indicates that the `reserve` rule is able to satisfy the reservation constraint of the `edit` rule. **Laputa** has two options: it may fire the rule now, or it may pre-fetch all of the files that are needed to fire the rule after the client has been disconnected. **Laputa** checks the type of action that is taken with the task, and if the action is `generated`, then it is performed prior to disconnection. Long duration tasks such as editing are typically `creative`, forcing the system to pre-fetch additional files to support the activity during disconnected operation. In this case, the `reserve` action is `generated` so **Laputa** reserves the file while there is still a connection to the central repository, reducing the number of overall files that need to be pre-fetched.

With the `edit` rule's condition satisfied, the system moves on to determines the activity type of the `edit` action. We see from figure 3 that the `edit` rule is a `creative` task, so **Laputa** pre-fetches the "C" source files onto the portable machine's local disk.

Once a user has finished `editing` a file, the effects of the `edit` rule are asserted, satisfying the conditions of the `reference` rule, which in turn forward chains to the `compile` rule. The possible forward chains are simulated before disconnecting from the shared repository, so **Laputa** can pre-fetch all files that are needed for these two tasks (the "C" source and its header files). The "C" file has already been pre-fetched for the `edit` rule, and is locked in `creative exclusive` mode, but the header files are only used as inputs to `generated` tools, so they are locked in `dirty read` mode. The system continues to explore its rule network and proceeds to pre-fetch those files that are needed for firing the `build` rule and the `test` rule.

Once file pre-fetching has concluded, the user disconnects the portable workstation from the network and is free to continue development. The user can edit "C" code, compile the files, build new executables, and test the code changes. All of the object code and executables produced during the disconnected period are locked in `generated exclusive` mode, and the source files are locked in `creative exclusive` mode.

When the user is ready to reintegrate, a network connection is reestablished. First, all of the modified "C" files that were locked in `creative exclusive` mode are copied from the **Laputa** client into the main repository. Then, all of the `dirty read` files are compared against the copies in the repository to check for differences. If none are found, then the object code and the executables locked in `generated exclusive` mode on the disconnected client are also copied into the main repository. If there are discrepancies between the repository and the files locked in `dirty read` mode, then all files which have dependencies upon the inconsistent files are deleted. Then the process is charged with regenerating all of the deleted files, in the central repository completing the reintegration.

7 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 over fifteen institutions to date.

8 Contributions

A related approach was taken in the Sun Network Software Environment [17]. 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.

Numerous other SDEs employ some form of checkout model for concurrency control, but we know of none besides **Laputa** that either exploits the software process to assist in selecting files to be checked out or that permits disconnected operation.

Acknowledgements

We would like to thank Dan Duchamp, who initially interested the authors in the problems of mobile computing.

References

[1] *2nd International Conference on the Software Process: Continuous Software Process Improvement*, Berlin, Germany, February 1993. IEEE Computer Society Press.

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

[3] 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.

[4] Israel Z. Ben-Shaul. Oz: A Decentralized Process Centered Environment. Technical Report CUCS-011-93, Columbia University, Department of Computer Science, April 1993. PhD Thesis Proposal.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, 39(4):447–459, April 1990.

[10] 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.

[11] 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.

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

[13] 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.

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

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

[16] Walter F. Tichy. RCS — a system for version control. *Software — Practice & Experience*, 15(7):637–654, July 1985.

[17] 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.