Ida Systems AB

Intelligent Distribution of Software Components to Thin Clients

Thesis

Project: ID	IDA-I-230	Name of document:	THES_COL
Document:	Thesis	Created:	1999-10-12
Responsible:	Magnus Jakobsson / Johan Söderqvist	Version:	0.6
Project manager:	Anders Olsson	Last revision:	2000-01-21
Customer:	Ulf Gundersen	Printed:	00-01-21 14:18

Abstract

Many software companies are migrating to server based component systems. Such distributed system uses thin clients. This gives lower administration costs, permits updates to be performed automatically and gives a facilitated deployment.

One of the companies that are adapting their software to this way of thinking is Ida Systems AB at Linköping. Most of Ida’s products are built on their component based platform PAX Enterprise, which now is to be transformed into a server based component system. However Ida wants to avoid the biggest drawback with thin clients, the waiting times experienced by users as components are downloaded from the server. Such perceived waiting times causes irritation and inefficiency and must therefore be minimised.

This thesis presents a solution to the waiting time problem in thin clients, by using intelligence at the server to model and predict users’ behaviour. Predictions are used to push components to a client cache before any explicit requests have been made. If a prediction is correct and the corresponding push operation finishes before the components are needed, the user will have the feeling that she uses a local program. In simulations based on PAX Enterprise usage it has been shown in this thesis that the proposed distribution model can cut down the waiting times by nearly 60%.

1 Introduction

This chapter gives a short introduction to this thesis and the examination project in general. It includes the project background, the purpose of the project, a description of the method of working used, the scope of the project and some general information about the document.

1.1 About the Thesis

This thesis is a part of the obligatory master’s degree examination project on the program “Computer Science and Engineering” at Linköping University. The work has been performed during the fall of 1999 at the company Ida Systems AB.

1.2 Project Background

Ida Systems AB (Ida) is a company in Linköping that specialises on document management systems and case management with workflow. These types of large-scale systems need to serve the user with information created in different applications on separate servers. Therefore the traditional client/server technology is inefficient and the programs have been based on a 3-tier architecture. This introduces a service layer between the servers and the clients, which offers the needed services and mediate data between them. Even though this solution results in less complex clients they still contain a lot of business logic, i.e. the clients are fat. This causes a lot of overhead when configuring and updating the systems since the administrators need to make individual installations for each user. By moving components from the clients to the servers Ida wants to create server based component systems. The purpose of this change is to avoid the problems described above and make future web adaptations easy.

If the migration is complete the result will be that all business logic is collected at the server-side of the system together with a repository for storing components. On the other side of the system resides client software that basically provides an infrastructure for downloaded components. These clients do not need any explicit business logic since this functionality is provided by the components. Such clients are called thin clients. By combining different sets of components at the server and sending them to the client, the user is presented with workspaces for solving specific tasks.

A drawback with the approach described above is that it causes a lot of network traffic, hence is likely to result in waiting times. To minimise this unpleasant effect the distribution mechanism could use personal information about a user’s everyday tasks and behaviour to make predictions about her next action. Based on such predictions, components can be pushed to a client before they are requested, thus hiding waiting times for the user and upholding the illusion of running a local program.

The scheme described above is heavily influenced by the one-to-one philosophy in today’s online marketing, see [Allen et al. 1998]. The idea with one-to-one marketing is to create a long-term individual connection with each customer, since it is much cheaper to increase sales to existing customers than to acquire new ones. Online shops track their users’ actions in order to create a knowledge base about individual needs and behaviour. The information is stored in profiles, used to dynamically create personalised web pages that contain products predicted to be especially interesting for the current viewer, personal rebates and support for earlier bought products.

In a server based component system with thin clients the users, just as in the case with customers in an online shop, must be kept satisfied. If they are delayed by the system that is supposed to make their tasks easier they probably will not use it, just as an unsatisfied customer is unlikely to return to a poorly designed online shop. The important question is, can personalised component distribution be used as effective as personalised web marketing?

1.3 The Target System

This part briefly describes the target system of the model presented in this thesis. Some fundamental expressions that are used in the rest of this thesis are also explained below.

Most of Ida’s products are built on their platform PAX Enterprise (PAX-E). This platform includes a presentation system and the possibility to integrate different building blocks, thus making it easy to build custom-made systems meeting the customer’s demand to effectively collect, spread, process and store information. The system is completed with client software and sold as a total solution to big organisations.

PAX-E includes workflow management. This means that Ida identifies the different work processes in an organisation and divides each of these into separate steps called activities. Similar processes and processes that depend on each other are grouped into cases. A case is opened when the first activity in the first process is started and closed when the last activity in the last process has been completed. To finish an activity a user must perform one or more tasks in the organisation’s PAX-E system. Usually the work is divided in such way that each activity is performed by a different user, making the case “travel” through the organisation. This makes it practical to associate the responsibilities in a project, denoted group in workflow management, with roles. Each user can belong to several groups and have many different roles. A role connects a user to certain activities and thereby certain tasks. In *Figure 1* an example of a process is shown.

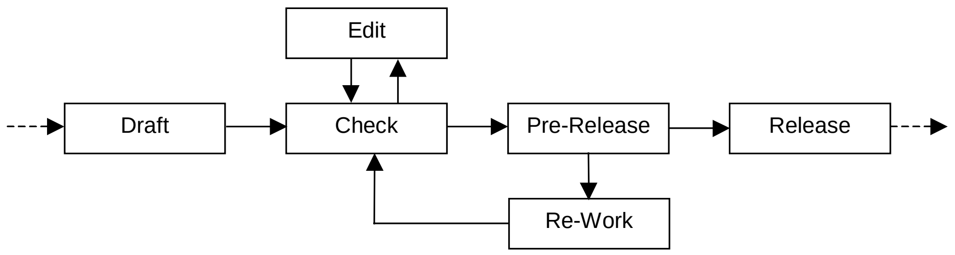


Figure 1. The activities in a workflow process defining the creation of a document.

The platform is designed with support for workflow in mind. Therefore it is divided into different workspaces where each of these present the components needed to perform one type of task. In *Figure 2*, a screenshot of PAX-E shows the concept of components and workspaces.

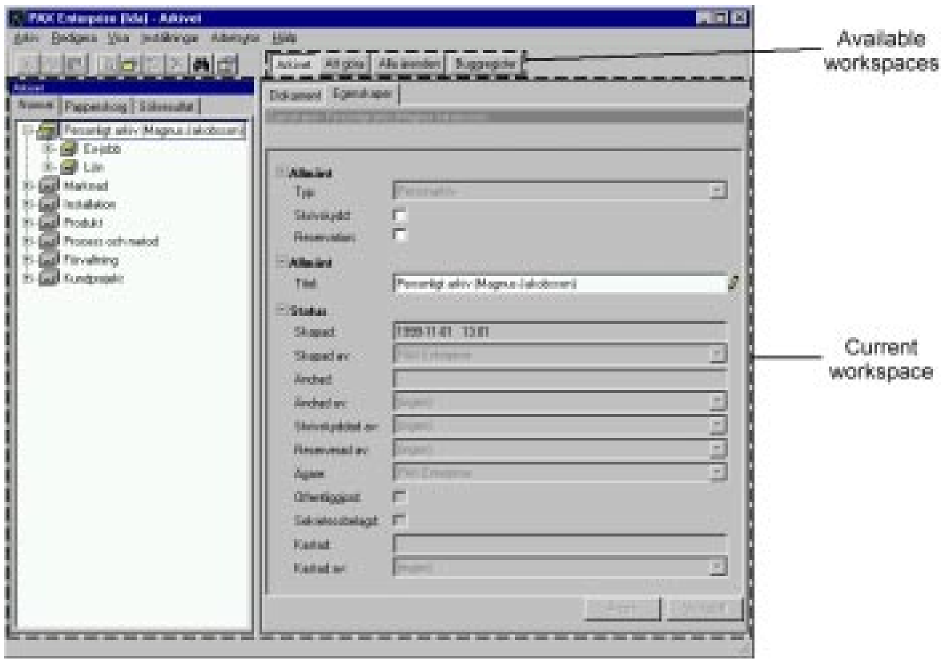


Figure 2. Screenshot of a PAX-E archive workspace with two components.

More information about PAX-E can be found in *4 PAX-E Observations*.

1.4 Purpose

The purpose of this examination project is to create a model for the distribution of software components to thin clients. This model should explain how user profiles, caching and automatic delivery of components is to be combined to solve the problem with waiting times in business applications using thin clients. The model should be adapted to work in PAX-E systems. Since the model is likely to be used in web based systems, platform independent aspects are to be considered.

An implementation in Java is to be made. It should consist of a server based component system with thin clients, where components are distributed as defined by the model. The implementation should show what level of performance that can be achieved by using the model.

1.5 Method of Working

The method of working used during the project time was a waterfall model with the following steps:

1) Studies of background theory to get a wide knowledge base. The material mainly consists of books, technical reports and online web pages.
2) PAX-E analysis. This includes working with the platform and interviewing people at Ida. The goal with the analysis is to find out how the targeted system can be combined with user prediction.
3) Modelling of the distribution mechanism. Different solutions and aspects are considered and then combined into a model.
4) Design and implementation of a testbench in Java. The purpose with this implementation is to create a way to evaluate and fine-tune the model.
5) Testing and fine-tuning of the implementation.
6) Conclusions and suggestions. How is the best solution, based on the model, configured? Does it seem to be worth the effort of integrating the solution in a business application or is the performance won negligible?

1.6 Scope

The examination project includes the following:

- The development of a distribution model for business applications.
- The implementation of a testbench.
- The writing of a thesis.

The examination project is limited in the following ways:

- The repository for storing the components is considered as given.
- The categorisation of the components and workspaces is also considered as given.
- No in-depth security analyses of the actual distribution are to be made.

1.7 Structure

This part gives an outline of this thesis. It also contains reading instructions, abbreviations and a glossary.

1.7.1 Outline

The rest of the document is disposed in the following way:

- Problem Definition. Contains a problem description and analyses of the different parts of the problem. The project requirements are also presented.
- Theoretical Background. Necessary background knowledge that is needed in order to understand the problem area and the solution presented in this thesis.
- PAX-E Observations. Summary of important observations made during the analysis of the target system, PAX-E.
- Distribution Model. Describes a distribution model, based on the theoretical background and the PAX-E observations, which solves the problem with waiting times in thin clients.
- Implementation. Presents the implementation of the distribution model in both functional and technical terms.
- Evaluation. Covers the testing of the implementation and shows how the distribution model performs in PAX-E simulations.
- Conclusions and Further Work. Presents conclusions based on the results shown in the evaluation. Also shows that the project requirements have been fulfilled and gives some ideas about further work.
- References. Complete list over material referenced in this thesis.

1.7.2 Reading Instructions

Here is some reading instructions that chapter by chapter describes which parts that are important to read and which parts that can be skipped.

In chapter 1 it is necessary to read about the project background and the purpose with the project.

Chapter 2 is essential to understand the problem that is to be solved.

A lot of the information presented in chapter 3 might be familiar to the user and can therefore be skipped to some extent. However it is very important to understand the following areas: server based components, user prediction, the network aspects discussed in the push introduction and client-side cache.

Chapter 4 covers target system observations, needed to understand some of the model and implementation choices.

Chapter 5 is essential to understand the proposed solution.

In chapter 6 it is important to study the technical architecture and what the implementation logs from the sessions.

Chapter 7 is essential to understand how the distribution model should be configured and what level of performance is to be expected in the target system.

In chapter 8 it is important to read about algorithm choice and the usage of the distribution model.

1.7.3 Abbreviations

The abbreviations used in this thesis are presented below.

CORBA	Common Object Request Broker Architecture
DAG	Directed Acyclic Graph
DCOM	Distributed Component Object Model
FIFO	First In First Out
GUI	Graphical User Interface
HTTP	Hyper Text Transfer Protocol
IDA	Ida Systems AB
IDL	Interface Definition Language
IPAM	Incremental Probabilistic Action Model
IUI	Intelligent User Interface
JDBC	Java DataBase Connectivity
LAN	Local Area Network
LFU	Least Frequently Used
LIFO	Last In First Out
LPC	Local Procedure Call
LRU	Least Recently Used
LRV	Least Relative Value
LZ	Lempel-Ziv
OMG	Object Management Group
ORB	Object Request Broker
PAX-E	PAX Enterprise
PAX-NG	PAX Next Generation
PPM	Prediction by Partial Match
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SKL	Statens Kriminaltekniska Laboratorium
TDAG	Temporal Directed Acyclic Graph
WSC	Workspace Selection Component

1.7.4 Glossary

Some of the terms used in this thesis are explained below.

3-tier Architecture	Advanced distributed computing model. This architecture introduces a service layer between the clients and the server. The usage of three tiers results in higher performance, better scalability and a more secure system.
Business Logic	The part of an application that performs the required data processing of the business. This includes the functionality that performs data entry, update, query and report processing. A server application is mostly business logic. A client application is made up of a graphical user interface and may also include some business logic.
Common Object Request Broker Architecture	A standard for communicating between distributed objects, possibly written in different programming languages and residing on different platforms.
Distributed Component Object Model	Extension of Microsoft's component object model providing a network protocol for running distributed objects. The technology is platform and operating system dependent.
Discrete Sequence Prediction	Uses a history, consisting of a sequence of atomic symbols that is ordered by the time of appearance, to predict which symbol is most likely to follow.
Fat Client	Denotes a client application that contains not only a graphical user interface but also a considerable amount of business logic.
Framework	The part of a thin client application that provides an infrastructure for plugged in lightweight components.
Heavyweight Component	A software component that contains business logic, a communication interface and possibly a graphical user interface.
Hit Ratio	Measurement of the performance of a cache. Calculated as the number of locally stored data requests divided by the total number of requests.
Incremental Probabilistic Action Modelling	A discrete sequence prediction algorithm that bases predictions on a table of previously seen symbols and the corresponding weighted probabilities.
Lightweight Component	A software component that contains a communication interface and a graphical user interface.
Lempel-Ziv	A discrete sequence prediction algorithm that bases predictions on a tree, consisting of previously identified patterns and corresponding frequencies of occurrence.
Markov	A discrete sequence prediction algorithm that bases predictions on a table of previously seen symbols and the corresponding frequencies of occurrence.

PAX Enterprise	Software platform made by Ida Systems AB. Used to create tailor-made document management systems and case management systems with workflow.
Prediction by Partial Match	A discrete sequence prediction algorithm that bases predictions on multiple tables of different dimensions, containing previously seen sequences of symbols and the corresponding frequencies of occurrence.
Prediction Ratio	Measurement of the performance of a discrete sequence prediction algorithm. Calculated as the number of correct predictions divided by the total number of predictions.
Profile	Software entity describing a user’s preferences, behaviour and privileges.
Push	Server initiated data transfer, automatically started before any explicit client request is received.
Remote Method Invocation	A platform independent built-in mechanism in Java for calling methods in remote objects located on other computers.
Server based Component System	Distributed system with framework clients and all business logic collected at the server. The business logic executes in application servers that the client queries by plugging in lightweight components, downloaded from server storage.
Software Component	An indivisible collection of code that solves a specific problem and has a well-defined communication interface.
Temporal Directed Acyclic Graph	A discrete sequence prediction algorithm that bases predictions on a tree with nodes corresponding to previously seen symbols and counters showing how many times a node has been visited and followed by one of its children.
Thin Client	Denotes a minimal client application that contains a graphical user interface without any business logic.

2 Problem Definition

In this chapter an overview description of the problem area is presented. An analysis of the area gives a deeper view of what needs to be done. The requirements of the project are also defined.

2.1 Problem Description

To be able to minimise the waiting times for the user running a thin client, components should be sent from the server before they are explicitly requested. Such pushing of information needs to be based on some sort of predictions.

How can one predict a user's next move? In [Davison & Hirsh 1998] it is asserted that humans tend to repeat themselves, making it possible to predict future actions by examining earlier behaviour. In a business application this would mean that users are likely to evolve certain working patterns when solving their daily tasks. If such patterns could be identified it should be possible to predict a user's future behaviour based on what operations she has performed in the past. Thus to be able to make predictions, information needs to be gathered about a user, workspace privileges and usage of components. This information can be stored in a personal profile. Such profiles can be kept on the server and used by a distribution mechanism to make decisions on what to send to an active client. The primary problems in this project are to decide what needs to be stored in the profiles and how the push decisions are to be made.

It is impossible to always make the correct prediction otherwise users would not be needed at all in today's computer systems. Due to this it is important to consider when it is effective to push components to the client and when it is better to wait for explicit requests. This implies that the push decisions need to be combined with a pushing policy that defines in which situations it is suitable to automatically send components.

The profiles needs to be dynamically updated, otherwise they could quickly become obsolete and give a false image of the user they model. How such incremental learning is to be performed is an important question that needs to be answered in this project.

For pushing to work the client must be able to store the automatically delivered components until they are actually used. Otherwise one must not only predict what the user will do next but also when the operation will be performed. Therefore the project also includes the creation of a client-side cache.

As can be seen, the model developed in this project must consider several different aspects of the distribution of components. Those aspects are explored more in section *2.2 Analysis*.

2.2 Analysis

Below follows a short summary of the different problems that need to be considered during the development of the model.

- What information needs to be stored in the profiles?
- How should the profiles be structured and updated?
- How are push decisions to be made?
- When should push be used?
- How are components to be locally stored at the clients?

These are analysed in the following subsections.

2.2.1 Profile Information

The purpose of the profiles is to minimise the number of components to choose among when pushing data to the client. One must compromise between the size and the effectiveness of the profile. A profile containing an extensive collection of information probably makes it easier to do a sound prediction, though it takes longer time to accomplish it. The size of the profile also affects the usability of the model in a more direct way. If the profiles require too much memory the scalability may be insufficient as the number of concurrent clients increases in a system. Another important aspect of the profile is how it affects the administration of the system. It is desirable that the profiles do not introduce extra maintenance work. To keep the clients as thin as possible the profiles should be stored on the server.

A study of today's different predictive systems, see *3.2.1 Today's Predictive Systems*, shows that a profile can contain the following types of data:

- Static data consisting of personal information.
- Dynamic data consisting of traces of the user's behaviour.
- Session data consisting of information that is typical for the current usage.

Examples of static data in the targeted system could be a user's roles and everyday tasks. Such information can be used to select a set of workspaces that a user should be able to access. Depending on what roles and privileges a user has in the system, different workspaces are available.

An example of dynamic data could be a continuously updated history over which workspaces and components a user has accessed. The dynamic data is the information that the push decisions primarily are based on. It should be stored in a data structure suited for the algorithm that makes the predictions.

Typical examples of session data are current bandwidth and preferred type of client software. This kind of data can be used to adapt workspaces for the machine running the client software. There could be several components, providing the same functionality, but to be used with different platforms.

2.2.2 Push Decisions

The distribution mechanism needs to make push decisions. These consist of two parts. First of all it must decide whether to automatically push a set of components or to wait for an explicit request. If the decision is to push, a prediction of what components to transfer is also needed. Since PAX-E is divided into workspaces consisting of components, the possible sets of components from which to choose among are already defined.

An important issue is how many future selections of workspaces that are to be predicted. There are two alternatives, either one restricts the prediction to one step ahead due to uncertainty or one thrusts that a prediction is correct and uses this to make another prediction of what is most probable to follow the earlier prediction. The second approach can be extended to allow predictions to cover a whole chain of future requests, but since it is impossible to always make correct predictions the length of this chain is limited in practice.

There are different solutions to how the decision making is to be done. One solution is to begin by predicting which workspace that seems to be best to push. This is done by examining the dynamic data in the user's profile. If a repetitive pattern matches the current situation, that is it begins with accesses to the same workspaces that the user now have made, the workspace that generally follows is chosen. After this the prediction is weighted and if it is better than a certain threshold, representing when push is to be used, it is realised. Another solution can be to use a static push policy in which all suitable situations for using push are defined. If the current situation belongs to this set a prediction is triggered and automatically pushed.

Some preferred characteristics to have in mind while designing the algorithm are:

- Speed. The algorithm must be fast enough to select and transfer a workspace before it is needed at the client. Time-consuming operations will neutralise the purpose of using push.
- Prediction quality. The algorithm has to achieve a fairly high level of accurate predictions. Otherwise it will fail to reduce the perceived waiting times at the client and also cause a lot of unnecessary network traffic.
- Manageable data structures. The algorithm should not use too complex data structures since the update operations need to be performed online. Incremental updating makes the algorithm dynamic and also lessens the server load during critical hours (i.e. in the morning and the afternoon when everyone logs in and out at approximately the same time).

The desired characteristics conflict with each other, making it necessary to make compromises. To summarise, the algorithm should be as effective as possible while still being suitable for online usage.

2.2.2.1 Initial Push Configuration

When a user uses the system for the first time there is no knowledge of previous behaviour to base predictions on. Before enough dynamic data is collected about a user an initial push configuration is needed to avoid a lot of bad predictions. This could be solved in different ways. Some of them are listed below.

- Do not use push until enough knowledge has been collected about a user.
- A new profile is initialised with dynamic data typical for the roles the user has. This requires explicit knowledge about which roles exist, what roles the targeted user has and what kind of behaviour each role represents.
- The first user belonging to a group of users automatically defines the typical behaviour for that group. In this solution the dynamic data collected during a learning period is saved and used to initialise all new similar users.

Since the idea with a server based component system is to avoid as much individual configuration as possible, the initial configuration of the component distribution should be automatic. If there is no role or group based solution that works automatically and gives good prediction directly from the start, it probably is best to avoid pushing until enough knowledge has been acquired.

2.2.3 Caching

For push to work some caching policy on the client-side is needed. However the usage of caches is also motivated by the possibility to gain performance by storing frequently used components locally. To avoid unnecessary file transfers some kind of extra communication between the server and the clients regarding locally stored components is needed.

The cache is supposed to be combined with the pushing solution to get a total solution that is as efficient as possible. A separate, standard caching solution will probably not perform well with pushing if it is not adapted to this kind of usage.

The size of the cache is another important issue. How many workspaces need to be locally stored at the same time to get enough performance? This depends heavily on how well the push solution works. It is positive if the cache can be kept small enough to be memory based since this permits faster access than in a disk based solution. There are also some situations in which a disk is not accessible, as in the case with normal Java applets.

Other forms of caching might be required to get scalability. There are several forms of proxy caching and distributed client caches that might solve scaling issues. By using such solutions the server can concentrate entirely on making predictions and pushing while explicit requests are handled at other levels.

2.3 Requirements

The requirements for the distribution model are:

- It is to be designed to work in a server based component system.
- It is to be adapted for PAX-E usage, however it is very positive if it is easy to generalise the solution and use it in other systems.
- It is not to increase the amount of administration work considerably.
- It should be easy to implement platform independently, permitting different types of clients.

3 Theoretical Background

This chapter present the theoretical background needed to fully understand the proposed distribution model. Distributed computing, user prediction, push technology and caching are discussed. All of the referenced material in this chapter is centred on similar usage of the technologies, making it suitable for further studies if the reader wants to get a deeper understanding.

3.1 Distributed Computing

Fast growing computer networks, like Internet, have widened the market for distributed systems. In [Coulouris et al. 1994] the following definition of a distributed system is given:

> *"We define a distributed system as a collection of autonomous computers linked by a network, with software designed to produce an integrated computer facility." [Coulouris et al. 1994 p. 1]*

The following main characteristics for a distributed system are described in [Coulouris et al. 1994]:

- Resource sharing. The resources shared by applications or users over the network can be hardware or data. This is the most fundamental concept of distributed systems.
- Openness. New units for sharing the resources can be added to the system without disruption of the currently available services.
- Concurrency. A mechanism for the resources and applications to co-ordinate and interact with each other.
- Scalability. There is no need for change in the system or application software when the scale of the system increases. Data can be replicated and the load can be shared among computers to increase scalability.
- Fault tolerance. To avoid failure in the system there are two approaches: hardware redundancy and software recovery. Hardware redundancy means that there is extra hardware that automatically replaces failed hardware. If software recovery is used the system automatically reconfigures itself to be able to continue if for example a connection breaks down.
- Transparency. Even though data may be processed at many computers at different places, the user of a distributed system should meet one single environment hiding the resources used to solve the task.

In *Figure 3* an example of a distributed system is shown. Computers are connected to a Local Area Network (LAN). Software supports the access to the resources, fileservers etc., connected to the network.

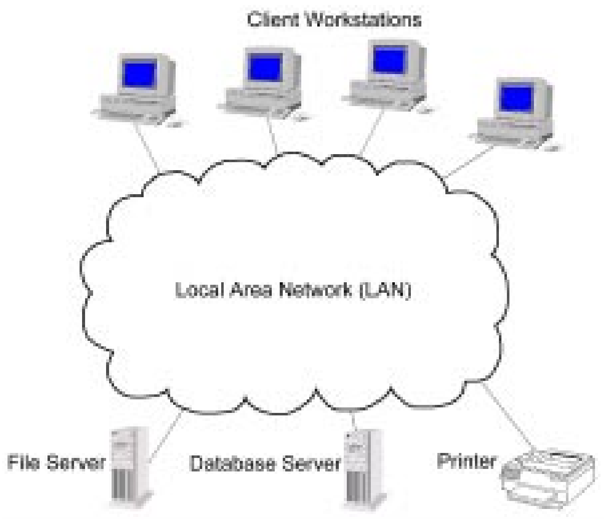


Figure 3. Typical parts of a distributed system.

In some cases a distributed system is a must. An example of this is if a company needs to access a customer database from all of the computers in a LAN. In other applications, like word processing, distribution is not suitable at all. There are several approaches toward distributed systems. Some of these are described in the following sections.

3.1.1 The Traditional Client/Server Model

In [Coulouris et al. 1994] the traditional client/server model is described. This model divides a distributed system into two parts. One part is the server that provides services. Examples of such services are database connections and access to hardware. The servers can be located at separate computers in the network and one service can be running in co-ordinated processes on different servers. The other part is the clients that use the services on the servers. Most of the business logic in these client/server applications is located at the clients. The term fat clients denote this.

In [d-tec 1998] two major drawbacks with the traditional systems are mentioned. The first is the fat clients, causing a lot of overhead during the deployment of a system and when a new release is to be installed. They also make it hard to configure large systems since some types of changes need to be done at each client installation. The second problem is related to scalability. Traditional systems perform well up to a certain number of clients. When this limit is reached it is very hard and expensive to add capability for more clients.

Despite the negative sides of the standard way to build distributed systems it is still very common. The reason for this is that it is relatively cheap and easy to create such systems. Why add complexity and cost if the targeted use of a system does not require many users and is not likely to grow into a larger product in the future?

3.1.2 3-tier Architecture

The 3-tier architecture differs from traditional 2-tier architecture by introducing a new layer between the clients and the servers. The new layer in the 3-tier architecture offers services to the clients and decreases the need of business logic in the clients. Usage of thinner clients result in a performance gain because large requests for data are traded against small requests for services.

The typical 3-tier system is divided into the three tiers as shown in *Figure 4*. The client-tier presents the data to the user and makes necessary calls to the application-server-tier, depending on the user interaction. All the business logic is placed in the application-server-tier. The data-storage-tier is used to handle and encapsulate data storage. By executing servers on multiple machines simultaneously and dividing the requests among them, both performance and scalability are increased. Since clients do not have direct access to databases in this architecture a much smaller number of connection licenses are needed, saving a lot of money. Detailed information about the 3-tier architecture can be found at [Kauffman 1997].

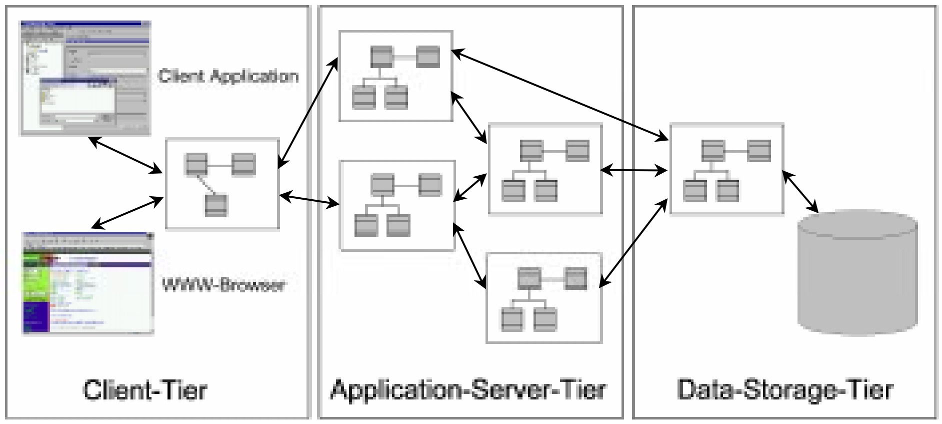


Figure 4. 3-tier system with two types of thin clients.

Moving the business logic from the clients to the server gives more advantages. The most important of them are presented in [d-tec 1998]:

- The user interface and the application logic are separated.
- Redesign of server services will not affect the clients.
- The security of transactions is higher.
- The change management is easier.

Since more of the logic is placed on the server the user interface could be made more independent of the server implementation. This increases the modularity of the system. If the server components need to be redesigned this does not affect the clients, since the service layer hides the server implementation. If high security is needed in the system this is easier to obtain with the 3-tier architecture model than with the traditional model because data can be processed at the service layer between the server and the client. Authentication when accessing the data storage is also easier to achieve because the only authentication necessary is the identification of a few application servers. When new versions of the software are developed less effort is needed to update the system. If the traditional client/server model with fat clients is used, all the clients have to be updated with the new software, but with the 3-tier approach a module can be changed on the server as long as its client interface remains the same.

Even though there are a vast number of advantages with a system implementing the 3-tier architecture using this solution needs careful consideration. It adds a lot of complexity to the system making it hard to design and implement. This results in a high initial cost. In other words, using a 3-tier architecture is a long-term solution that is justified if the system is expected to live long and have a large number of clients. *Figure 5* illustrates a general rule of thumb used in development of distributed systems.

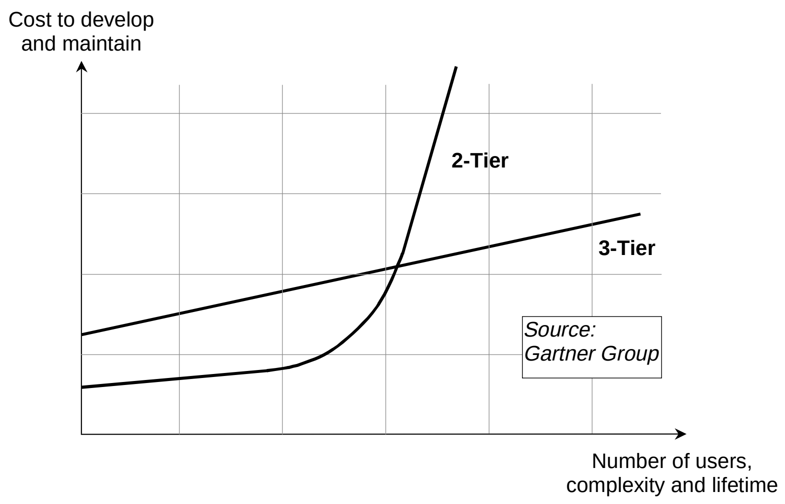


Figure 5. Cost comparison between 2-tier and 3-tier architecture.

3.1.3 Server Based Components

One of the biggest trends in distributed computing today is to migrate to server based component systems. These are centred on the usage of software components and combine this approach with the concepts of 3-tier architectures.

Components are an indivisible collection of code that solve a specific problem. They have a well-defined interface describing the provided functionality. It is most common that a component consists of one or more classes, although it could also contain functional procedures and global variables. There are a lot of opinions on what is and what is not a component. Clemens Szyperski gives the following definition:

> *"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties." [Szyperski 1998 p. 34]*

Server based component systems extend the 3-tier architecture in the following way:

- They are totally component based and all components are stored in a database on the server, usually called the repository.
- The system contains two types of components, heavy- and lightweight.
- All business logic is implemented as heavyweight components running on one or more application servers (in the application-server-tier).

- Lightweight components present a graphical user interface at the client, which can be used to access the corresponding business logic on the server.
- The software on the client only provides an infrastructure for lightweight components.
- When a task is to be solved by a client the necessary lightweight components are requested and downloaded from the repository on the server. After this they are deployed and activated, permitting the user to perform the task.

The discussed parts of a server based component system are shown in *Figure* 6.

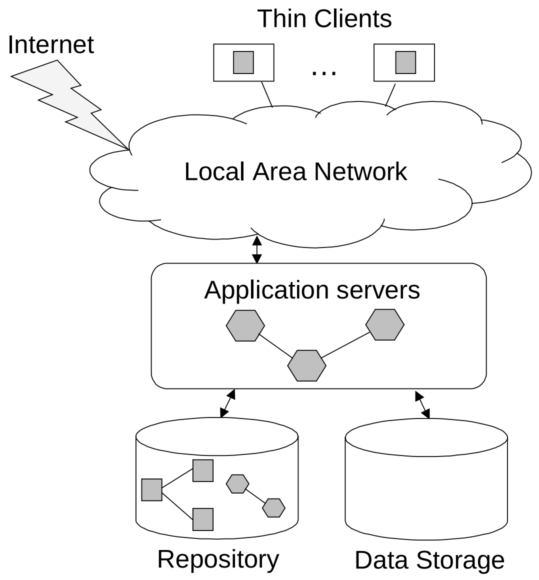


Figure 6. A typical server based component system.

In a server based component system the size of the clients is minimal and all configuration and versioning can be managed from the server. This saves a lot of administration costs. One also gets more efficient development since components and designs can be shared and reused.

It is very likely that the market of third-party components will grow rapidly in the near future, making it possible to buy and integrate high quality code with your own business applications.

Since this model is relatively new, most organisations that start to use this have to change their development strategy to fit the component thinking. Learning to create and co-ordinate complex components takes a lot of effort. Another drawback with server based components is that it can generate heavy network traffic.

There are a lot of information about server based component systems on the web, a good place for deeper studies is [Sun 1999/4].

3.1.4 Different Technologies

The three main solutions available today for using and co-ordinating distributed components are Remote Method Invocation (RMI), Common Object Request Broker Architecture (CORBA) and Distributed Component Object Model (DCOM). In this section follows a short presentation of each technique and a performance comparison.

3.1.4.1 Remote Method Invocation

Sun's Java supplies a hidden transport layer that handles data encoding, transmission and call protocols. This is used by the built-in mechanism RMI for calling methods in remote objects, located on other computers. For this to work a client must be able to locate a remote object. A remote RMI registry on the server provides this service.

When a client object wants to make use of a remote object's method it calls a regular method encapsulated in a surrogate object called stub. Stubs reside on the client and, among other things, use the remote RMI registry to get references to remote objects. When the stub has performed the lookup it packages all parameters and uses the transport layer to send an invocation to the server. On the server a corresponding object called a skeleton resides. This is the object that the stub finds in the remote RMI registry, provided that the skeleton has registered itself before the lookup takes place. The skeleton automatically receives requests and extracts the information in them. After this it passes on the information to the actual object providing that remote method. Finally it returns the result to the stub, which passes on this information to the object that initiated the request. The returned result can be a simple data structure or a complex object. The described course of events is illustrated in *Figure 7*.

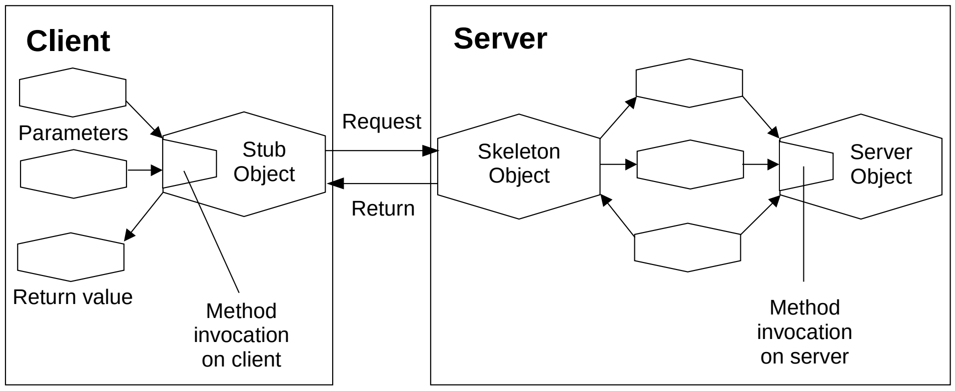


Figure 7. The principal of a remote method invocation in Java.

To create the needed stub and skeleton the programmer creates a Java interface and sends it through a Java tool that automatically creates the needed classes. Hence there is not much effort needed from the programmer.

As can be seen RMI is an extension of Java and therefore only supports this language. Although this is compensated for since Java is platform independent. The advantage with RMI is that it is well-integrated and easy to use, almost nothing needs to be done outside the Java environment. More information regarding RMI can be found at [Sun 1999/1].

3.1.4.2 Common Object Request Broker Architecture

The Object Management Group (OMG), an independent consortium, provides another solution to the need of co-ordinating distributed components. This solution is called CORBA. It is a specification and not an implemented product. There are many implementations to choose from.

The programming model in CORBA is similar to the one found in RMI. Since CORBA is language independent it is more complex. In *Figure 8* the most important parts of CORBA are illustrated.

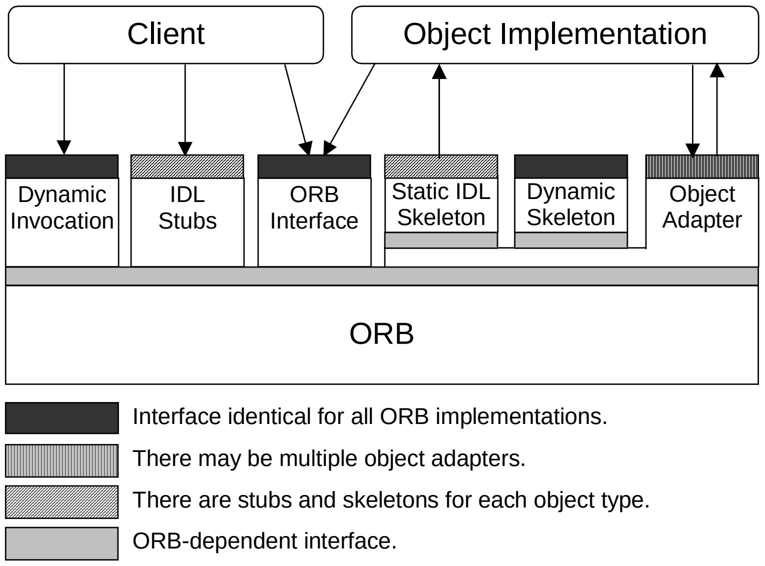


Figure 8. The fundamental architecture of CORBA.

To begin with there are two ways for a client to invoke a remote object’s method, either by using a stub or by using a dynamic invocation interface. The stubs are specified in an independent language called OMG Interface Definition Language (IDL). These files are then passed through language specific IDL compilers that create the corresponding classes in the target language. Stubs are static, if one wants to make invocations “on-the-fly” the standard dynamic invocation interface can be used instead. With this interface a client object can learn about the server object’s services and create an appropriate call. Thus this interface is independent of the actual target object’s interface. No matter which way of invocation is used the request will be sent via the Object Request Broker (ORB). The ORB provides the needed infrastructure and some common services. When a request reaches the ORB it uses its naming service to find the remote object implementing the requested method. After this it uses either an IDL skeleton or a dynamic skeleton, depending on the type of request, to forward the invocation. The result is sent back in the same manner. Objects on the client and the server can use an ORB interface to communicate directly with the ORB. This interface gives access to a few special services. Finally there is an object adapter between the ORB and the server objects. Examples on services provided by object adapters are generation and interpretation of object references, mapping and registration, object activation and deactivation, method invocation and security interaction.

If the communicating objects reside on different ORBs the Internet Inter-Orb Protocol (IIOP) can be used to connect them. This introduces another layer in the CORBA model that connects different ORBs over Internet.

The advantage of CORBA is that it is entirely language and platform independent. The disadvantage is that it is very complex and takes time to understand and use. It also requires additional programming knowledge since IDL is used to create the necessary stubs and skeletons. This summary of CORBA is relatively brief, for more information and specifications see [OMG 1999].

3.1.4.3 Distributed Component Object Model

The third technique is Microsoft's DCOM, described at [Microsoft 1997]. It is an extension of the Component Object Model (COM) where a network protocol replaces local interprocess communication. DCOM is language independent but platform and operating system dependent. The services provided are almost the same as in CORBA. A confusing difference is that stubs are called proxies and skeletons are called stubs.

If a client wants to call a remote method it makes a request to DCOM. This will cause DCOM to load the correct proxy into the client's address space. DCOM will also load a corresponding stub in the server's address space and make a reference to it in the proxy. The proxy and the stub handle the communication automatically using DCOM, in much the same way as in the case with stubs and skeletons in RMI and CORBA. To be able to know when a DCOM generated object is not used anymore DCOM uses reference counters, when such a counter reaches zero the corresponding object is unloaded.

In DCOM there is also a mechanism for making dynamic invocations. It is called automation. With an automation object at the server, a client can learn about its services during execution and build an appropriate call without using static information.

One extra feature that DCOM has compared to the other two techniques is that it supports inter-process communication on the same machine with a lightweight mechanism called Local Procedure Call (LPC). Otherwise the more complex Remote Procedure Call (RPC) is used.

DCOM's advantages are its language independence and the LPC mechanism, but the platform and operating system dependence is a major drawback. And though it supports Java it requires the Java Virtual Machine to execute on one of the supported platforms.

3.1.4.4 Performance Comparison

In this section two independent performance tests of the different techniques are presented. This will give a feeling of how fast they are and how much effort that is required from the programmer when they are to be used with Java. In both of the tests the CORBA implementation used is VisiBroker from Visigenic.

In [Edlund 1998] a comparison of all of the above techniques is made regarding the speed of server calls and data transfer from the server to the client. In this comparison both the client and the server is implemented in Java and the data sent are arrays of objects. Two test configurations have been used; one with the server and the client located at the same computer and another with them located at different computers. The main results in the performance tests are listed below.

- RMI has by far the longest waiting times for sending the data used in this test. It does not matter whether the client and the server run on different computers or not.
- DCOM is the fastest technique if the client and the server are located at the same computer. The reason for this is the LPC feature.
- CORBA is slightly faster than DCOM if the client and the server are located at different computers.

In [Edlund 1998] further results show that RMI got a big advantage over the other techniques when the programming effort is measured. The reason for this is the tight integration with the Java environment. CORBA is not as easy to use as RMI since the programmer need to use the external OMG IDL among other things, but it works well with Java. It also tends to increase the complexity of the application. DCOM and Java are not a good combination. The integration is bad, causing serious limitations and problems.

In [Juric et al. 1998] RMI's and CORBA's invocation times are compared with each other. The performance test was extensive using multiple testing patterns and different data types of parameters and return values. The program used in the tests was an automatic teller machine application implemented in Java and there was different tests ranging from one up to eight clients. For a more detailed description of the test see [Juric et al. 1998]. The main results are:

- On fast computers the RMI performance is acceptable, sometimes even comparable to CORBA, provided that the data amounts sent between the objects are small.
- Under heavy client load CORBA outperforms RMI. With eight clients CORBA was more than twice as fast as RMI.

The conclusion from these tests is that if Java is to be used the choice stands between RMI and CORBA. If the system is big and fairly complex, as in the case with a commercial system using component distribution, CORBA is preferable. In smaller systems RMI might be a worthy contender. The biggest reason for this is that it is so much easier to use RMI and that the introduced complexity with CORBA and its many layers can cause a performance degrade in small systems.

It should be noted that as this thesis is written Sun and IBM have released RMI over IIOP. This introduces fully CORBA compliant capabilities in Java. This solution gives the programmer the ability to use both RMI's simplicity and CORBA's complexity. More information about this can be found on [Sun 1999/2].

3.2 User Prediction

Users act in different ways and their actions are not always easy to predict. However by focusing on a single user or a group of similar users, patterns are often likely to occur either with or without consciousness, see [Davison & Hirsh 1998].

It seems like repetitive behaviour is part of human's nature. An example of this can be a person's morning routine. It is not entirely uncommon that it goes something like this: she wakes up and takes a shower, clothes, makes breakfast, eats, brushes her teeth and goes to work. Of course this order is broken every once in a while, but most of the mornings it is the correct procedure. Based on the fact that it is morning and this person has taken a shower and clothed, the average prediction that she will enter the kitchen next is very likely to be fulfilled. This knowledge could be used to create automated electrical home appliances products that for example have a glass of fresh orange juice ready when the person is finished showering, provided that she has followed the procedure so far.

Exactly the same kind of repetitive patterns tend to appear when a person uses a business application at work, i.e. after saving a new contact in the customer database the user usually opens a workspace for entering personal notes about this customer's preferences.

Correctly used user prediction can make our lives easier in many ways. This chapter presents computer systems utilising prediction, important design issues in such systems and common approaches (including data structures and algorithms).

3.2.1 Today's Predictive Systems

Models for predicting user actions are used in several areas today. Most of the systems are made for research purposes. Some of these areas are presented in the following subsections.

3.2.1.1 Command Prediction

The purpose with command prediction is to make command shells that try to predict what command the user will input next. It has been shows from studies of command histories that a user's sequence of commands contains a lot of patterns. With a history of the current session and knowledge of commonly repeated interaction sequences it is possible to predict the next input. The knowledge can be acquired by studying data collected from the user's previous sessions. The usage of patterns can be used together with presence frequencies of the different commands.

The user modelling in command prediction systems are usually extensions of traditional machine learning algorithms. Most of these build some kind of probabilistic graph, which is used to predict the current user's next input based on the history.

There is a lot of research in command prediction and numerous reports present algorithms and different kinds of implementations. In [Davison & Hirsh 1997] and [Davison & Hirsh 1998] extensive analysis of this field can be found.

3.2.1.2 Intelligent User Interfaces

According to [Lau 1999] the increasing use of complex computer systems opens up the market for applications supporting the user. Non-expert users may need to be guided through the system whereas expert users may need support to speed up their tasks. By predicting future user actions, user interfaces can be made adaptive to the user's needs and make her work easier. These interfaces are called Intelligent User Interfaces (IUI). They are used to increase a system's usability.

Examples on different uses of IUI are intelligent tutoring systems, natural language systems and systems with adaptive graphical user interfaces. One of the most interesting possibilities with IUI is to create adaptive help systems where answers are based on the current user's knowledge and experience. Information about such a system can be found in [Bladh 1995]. As in the case with command prediction, systems with IUI usually rely on machine learning algorithms.

3.2.1.3 Information Filtering

The usage of information filtering based on user models is getting rather common. The essence of information filtering is to find and present a set of relevant information out of the superset of available information.

There are different areas of application of information filtering such as: to help users search web pages, push relevant web pages via web channels, help users find relevant information in a database system and to support users in document handling systems.

A common solution in information filtering systems is autonomous software agents. The user models used by these agents are usually based on a collection of keywords, weighted by a factor of relevance [Çentimentel et al. 1999]. The agents use these weights together with the frequency of appearance of the keywords to decide if a document is relevant. To learn which documents that the user found interesting explicit feedback often is needed.

3.2.1.4 Similarities Between the Areas

As can be seen above the different areas where user prediction is utilised share similarities in their approaches. The reasons for this are that all need to use some kind of user models and that there exists a common goal to create an individualised system helping the user become more efficient. The result of the similarities is that the algorithms used for user modelling and prediction can be used in many different kinds of applications. This is of course not entirely true in practice since many of the algorithms are optimised for usage in a certain area.

In the field of command prediction and IUI it is very common to make user predictions by using discrete sequence prediction. Philip Lard and Ronald Saul define discrete sequence prediction in the following way:

> *"In discrete sequence prediction, the input is an infinite stream of atomic symbols. The task is to find statistical regularities in the input so that the ability to predict the next symbol progresses beyond random guessing."*
> *[Laird & Saul 1994 p. 43]*

There are three requirements that a system must fulfil in order to be suitable for usage with discrete sequence prediction algorithms:

1) The history of a user's behaviour must be divisible into atomic actions.
2) The history of a user's behaviour must contain repetitive patterns of such atomic actions.
3) There must not be a large quantity of different atomic actions, if there are more than a hundred different atomic actions most algorithms fail.

Provided that a system can be created so that these requirements are fulfilled, it is very likely that discrete sequence prediction can be used to model its users.

3.2.2 Design Issues

The first thing one must consider before developing a system using user prediction is the question: is there a need for user models in this system? The approach with user models and prediction mechanisms is not always suitable. Even if the target system fulfils the requirements mentioned in *3.2.1.4 Similarities Between the Areas* it may not be a good idea to use user prediction. The reason for this is the complexity that this feature tends to add to a system. Making user prediction work is difficult and requires considerable efforts.

In [Bladh 1995] user modelling in a help system is analysed. [Bladh 1995] mentions a few general issues one must consider when developing an adaptive system, these are:

- Should the user model be generic or individual? If the users can be divided into groups with similar needs it could be easier to design generic user models based on these groups.
- Should the user model be static or dynamic? It is much easier to create a system whose user models are static. However static modelling causes serious limitation due to the fact that people tends to change their behaviour with time as they get new working tasks.
- Should the user model be long- or short-term? Long-term user models make use of knowledge that is not only useful in the current interaction. Short-term user models specialise on a small part of the usage of the system. The choice between these entirely depends on the area of application.
- Should the user model collect knowledge implicitly or explicitly? In some systems it is important to get explicit information from the user to be able to make an accurate user model. Of course one can combine the usage of explicit knowledge with implicit knowledge acquisition.
- Should the user model be visible or hidden? In a visible user model the user can inspect the contents of the corresponding model. This can build a good relationship between the user and the prediction system. If a system with visible user models is extended with functionality permitting the users to control and fine-tune the models it could get more effective.

3.2.3 Common Approaches

There are many different approaches to how the problem with user prediction should be solved. In this section some common data structures and algorithms that seem to be applicable in the area of component distribution are presented. These have been selected from the field of machine learning with discrete sequence prediction in mind, see *3.2.1.4 Similarities Between the Areas*.

3.2.3.1 Fundamental Data Structures

In user prediction some kind of storage is needed to keep the data that represents knowledge about the user’s behaviour. The main purpose of such structures is to be a basis for the predictions.

In discrete sequence prediction most algorithms rely on extensions of Directed Acyclic Graphs (DAG), denoted probabilistic graphs. In these graphs there are nodes representing different states of the system and directed connections between them symbolising possible transitions. Each transition has attributes that represent the probability of its occurrence.

With probabilistic graphs it is only possible to take the current state of the system into consideration when making a prediction. In many cases it is desirable to not only use the current state but also information about the path to it. This can be done with N-dimensional arrays. In such data structures the *N-1* transitions before the current state are also used to predict the next state.

3.2.3.2 The Markov Model

The Markov model is described in [Russell & Norvig 1995]. This model is widely used in many different fields of artificial intelligence, for example speech recognition and decision making. Several algorithms for command prediction are based on the Markov model. The model describes a process and it is based on a number of states with transitions between them, see *Figure 9*. A probability value is assigned to each of the transitions. In this model the next state is only depending on the current state. There are no dependencies of previous states. It is easy to implement the Markov model using a probabilistic graph. In most cases this results in a Markov tree.

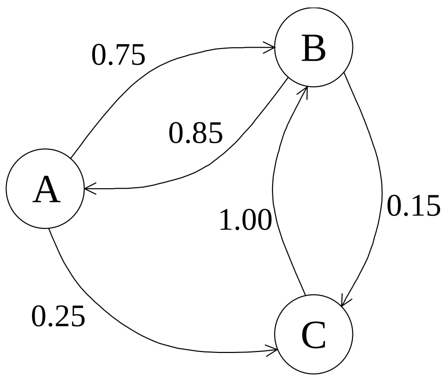


Figure 9. Example of a Markov model with three states.

The Markov algorithm can be extended to use a finite number of elements in the history. The depth of the history varies with different applications [Lau 1999]. This is called the Markov model of N^{th} order if the predictions are based on the most recent *N* steps of the history.

The following example from [Curewitz et al. 1993] explains a second order Markov prediction. Let the input sequence be *ABAABAABBAAB*. Since the prediction is of second order it is based on the last two items in the sequence *AB*. In the input sequence the block *AB* is two times followed by *A* and one time followed by *B*. Therefore the probability of *A* as the next item in the sequence is 2/3 ≈ 66% and the probability of *B* is 1/3 ≈ 33%.

A drawback with the traditional Markov model is that its precision is limited. However there exist numerous extensions of this model where the precision has been improved, examples of such follow below.

3.2.3.3 Temporal Directed Acyclic Graph

Temporal Directed Acyclic Graph (TDAG) is an extension of Markov trees [Laird & Saul 1994]. This algorithm is constructed with important parameters, like time and space required for generating a prediction, in mind. The basic principle of TDAG is to construct a tree, where each node corresponds to one discrete symbol in a sequence of input.

Every node has two counters: `in-count`, which tracks the number of times the node has been recognised in the input sequence, and `out-count`, which stores the total number of times the node has been succeeded of one of its children nodes. A queue of nodes, `state`, is needed to store the nodes from which to continue with the updating of the tree. The `state` variable is initialised with a root node. If a node exists in the current `state` the probability of one of its children being visited next is calculated as the `in-count` of the child divided by the `out-count` of the parent. The modules of a simplified TDAG are presented in *Figure 10*.

```
input(x) /* x = the next input symbol */
  •  Initialise new-state := {Λ}
  •  For each node v in state,
       –  Let μ := make-child(v,x).
       –  Enquque μ onto new-state.
  •  state := new-state.
```

```
make-child(v, x) /* Create or update the child of v labelled x */
  •  Find or create the node μ with a symbol of x in the list of children(v). If
     creating it initialise both its count fields to zero.
  •  Increment in-count(μ) and out-count(v) each by one.
```

```
project-from(v) /* Return a probability distribution */
  •  Initialise projection:={}.
  •  For each child μ in children(v), add the pair[symbol(μ), in-
     count(μ)/out-count(v)] to the projection.
  •  Return projection.
```

Figure 10. Basic TDAG algorithm in meta-code. The input module is invoked every time a new symbol is input. The project-from module returns a probability distribution, which can be used to make predictions.

Using the algorithm to build the TDAG tree for the simple input sequence *A* followed by *B* gives the following example:

1) When the first input (*A*) arrives, a corresponding child node of the root node is created. The `in-count` of node *A* is set to one. The `out-count` of the root node is increased by one. The new `state` consists of the root node and the *A*-node.
2) When the second input (*B*) arrives, corresponding child nodes are created to the nodes in the `state`. The `in-count` of both *B*-nodes is set to one. The `out-count` for both the root node and the *A*-node is increased by one. The new `state` consists of the root node and the *B*-nodes.

The example results in the tree in *Figure 11*.

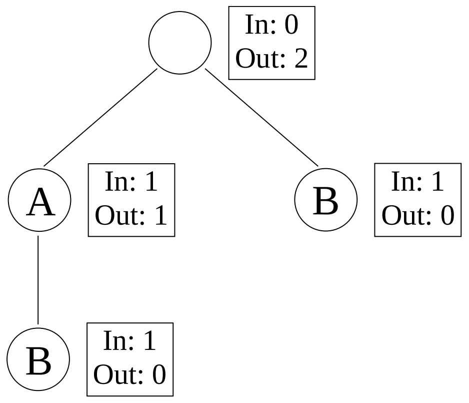


Figure 11. The tree that TDAG constructs to represent the sequence AB.

To predict the next input by using the sequence seen so far, the function `project-from` in FIGUR is used. This gives a distribution with the probability for each of the possible symbols. The input with the highest probability is chosen as the prediction.

The TDAG presented above was simplified. To use this algorithm in practice, some restrictions are necessary to limit the computations. The following limitations are bounded to reach this:

- The node probability. A node that represents a symbol that does not occur frequently is removed from the tree. A threshold value for the probabilities is introduced to select what nodes to remove.
- What node to trust. A node with a very low `out-count` should not be trusted when making predictions since the probabilities are very uncertain. A threshold value is introduced, defining the required `out-count` for a node in the `state` to be used when making predictions.
- The height of the tree. The height of the tree must be bounded in some way otherwise the tree will grow unlimited. A threshold value is introduced for the maximal height of the tree and nodes are not added if the threshold is exceeded.
- The size of the prediction. The number of distinct symbols is unknown and can be very large. Therefore some restrictions are wanted when using TDAG in real-time applications. This is solved by limiting the number of answers in the projection to the ones with the highest probabilities.

Which of these limitations to use and how to use them must be decided from case to case. Generally an `out-count` threshold and a height limit are required. The other limitations are mainly needed in situations where there are a large number of different symbols.

Some examples of applications where the TDAG algorithm has been successfully used are text compression, dynamic optimisation and predictive caching. To get the most out of this algorithm in a multi-user system a TDAG should be made for each of the users or for each group of users. This is due to the differences in behaviour and tasks of users.

3.2.3.4 Incremental Probabilistic Action Modelling

In [Davison & Hirsh 1998] the Markov model is extended to form an algorithm called Incremental Probabilistic Action Modelling (IPAM). This algorithm was invented for command prediction. IPAM is started with an empty table for storing data. Each new command given by the user causes the addition of a new row to the table. One essential parameter for this algorithm is α, which determines how much the prediction should depend on the history. With an α-value of 0 the history is disabled and the most recently used command is predicted. If α equals to 1 the probabilities are never updated.

The algorithm was evaluated in [Davison & Hirsh 1998]. The evaluation was made in a command prediction for UNIX. Data was collected from 77 users during a period of two to six months. Empirical tests gave the optimal value .80 of α in this environment.

The algorithm consists of the following steps for updating the probabilities in the table:

1) The user enters command c_1.
2) A new row for c_1 with a uniform statistical distribution is added to the table.
3) The user enters command c_2.
4) A new row for c_2 with a uniform statistical distribution is added to the table. The row for the previous command, c_1, is updated. This is done by multiplying each probability in the c_1-row by α and then increasing the probability of c_2 in the c_1-row by $1-\alpha$.

For each newly entered command step four is repeated and if the entered command already exists in the table the row of the previous command is updated. Checking the row of the current command and picking the one with the highest probability makes the prediction in the IPAM algorithm. IPAM can easily be adapted for usage in a server based component system by predicting workspaces instead of commands.

As shown above, the algorithm only makes use of the two most recently entered commands when updating the table. This is a drawback since the algorithm will not recognise long-term patterns in the command sequences.

3.2.3.5 Lempel-Ziv

The Lempel-Ziv Algorithm (LZ) is an algorithm designed for data compression. In [Curewitz et al. 1993] a character based LZ is extended to be used in the area of prefetching instead.

The basic principle of the LZ algorithm is to divide the input into pattern based blocks. The whole input sequence of items is divided into shorter sequences of length n. Each of these sequences is divided into blocks $x_0=\lambda, x_1, x_2, \ldots, x_c$ in such way that for all $j \geq 1$ every block x_j without the last item is equal to some block x_i for $0 \leq i < j$. The block λ represents the empty block. A probabilistic model can be built from the occurrence of patterns in these blocks. This model can be represented as a tree with weights on the branches. A weight can be implemented as an integer count of the number of visits to the node.

The following example explains the algorithm. With the premise that the input sequence consists of objects from the domain *{A, B, C}*, the input sequence is given as *AAAABACABCBAC*. The Lempel-Ziv decoder parses this as the blocks *A*, *AA*, *AB*, *AC*, *ABC*, *B* and *AC*. When these blocks are found, a probability tree can be built by considering the occurrence of these blocks. There are six blocks starting with item *A* and one starting with item *B*. Therefore the probability of branch *A* is *6/7 ≈ 86 %* which implies the probability of *1/7 ≈ 14 %* of branch *B*. If the first item is *A* the probability of the branches from this node will be *1/6 ≈ 17 %* for *A*, *2/6 ≈ 33 %* for *B*, and so on. When a pattern ends at node that is not a leaf a end-of-pattern leaf must be inserted to represent the probability of this. The complete probability tree is shown in *Figure 12*.

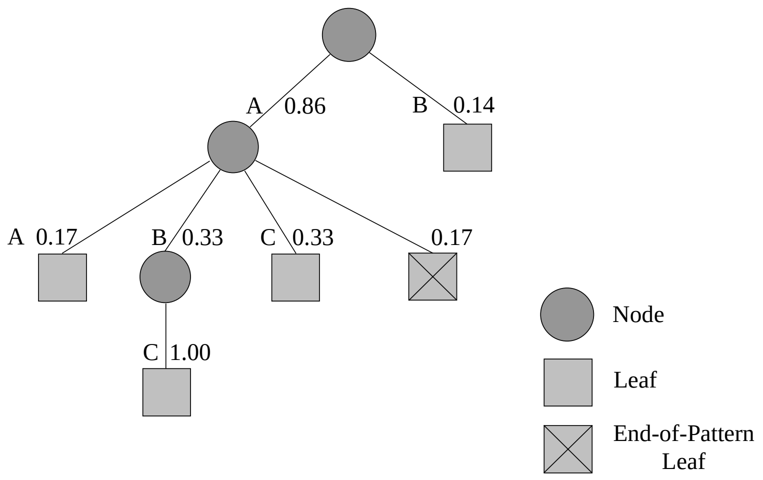


Figure 12. The tree LZ constructs to represent the sequence AAAABACABCBAC.

The resulting tree is used to make the predictions. By starting in the root node and choosing the item with the highest probability a prediction of the next item can be made. If the prediction is wrong when the actual request arrives the tree is updated to better reflect reality. If a leaf is reached the current pattern has ended and the traversal restarts from the root node. After *n* input items the algorithm will discard the tree and start building a new one. This must be done since the algorithm does not add new nodes to the tree otherwise, causing it to miss previously unidentified patterns in the input.

Even though the LZ algorithm is invented for data compression it seems to work rather well for prediction. Updating of the tree is simple and can be done during the traversal of the tree, since the weights are implemented as integer counters on each branch in the tree. One drawback with LZ is that the tree occasionally needs to be rebuilt. This generates a lot of unwanted calculation load on the server. Due to the dividing into blocks the LZ algorithm will not recognise all of the patterns in the input sequence.

3.2.3.6 Prediction by Partial Match

Another algorithm used for compression of data is the Prediction by Partial Match (PPM) algorithm [Curewitz et al. 1993]. This algorithm is rather simple and it is based on the Nth order Markov model, see section *3.2.3.2 The Markov Model*.

An Mth order PPM algorithm uses several Markov models of order N, where $0 \leq N \leq M$. These models can be combined in two different ways when making predictions:

1) Each individual model makes predictions. The different probabilities are multiplied by weights, giving preference to predictions of higher order since these are generally more reliable. Finally the prediction with the highest combined value is chosen. This results in that the faster models of lower order are used when the other models can not find a long pattern matching the current situation or has not been fed with enough data yet.
2) The currently chosen model only makes predictions. The order of the model to use when making a prediction is based on how well the algorithm has performed earlier. When a prediction is correct the currently preferred order is increased, permitting the algorithm to use longer patterns when making predictions. When a prediction is wrong the algorithm punishes itself by lowering the preferred order, thus limiting itself to use shorter patterns when making predictions.

Which way the PPM should be configured depends on the field of application. If it is known that shorter patterns are more common than longer patterns the second approach might be more suitable with a low value of M. Otherwise the first approach probably works better together with a high value of M.

The conclusions about using compression algorithms for prediction presented in [Curewitz et al. 1993] are that an algorithm’s prediction effectiveness is directly depending on its ability to compress data effectively. The PPM algorithm is found to be a bit more effective than LZ for both compression and prediction.

3.3 Push Technology

The term push technology was one of the biggest buzzwords in 1996. Since the push-revolution on the Internet failed, because of unfulfilled promises, most software companies have tried to avoid it in their marketing. This has resulted in a bad reputation and the widely used term “shove technology”. In this section the general ideas behind automatic delivery of data are presented among some different fields where it has been used with varying success.

3.3.1 Introduction to Push

The main idea of push technology is to automatically deliver data from a server to a client. Provided that the selected data is relevant and will be used in a near future, an explicit request has been avoided combined with the sensation that the download seems to be instantaneous at the client-side.

Important aspects that one need to consider before including support for push in a system are the high demands that it makes on computing resources and network bandwidth. The server not only needs to take care of the requests but also needs to make predictions and initiate transfers of the chosen data. Nevertheless hardware gets faster every day and it is also relatively cheap compared to software licenses. The most obvious drawback is the potential to waste bandwidth. If the predictions are faulty, unnecessary and useless traffic has been caused. Thus the quality of the predictions need to be fairly high. Another bandwidth related issue is that during peaks of high network utilisation the performance will degrade because of added background load. This may get so bad that, if the periods last long enough, the degradation will be noticed by the users. Also if there are time-critical sessions between certain hours the usage of pushing may have to be limited.

In [Crovella & Barford 1997] the network effects of prefetching are discussed in depth. Prefetching is based on the same idea as pushing but in this case the mechanism for automatic transfers lies on the client-side instead. Since the two techniques are so close, most conclusions apply to pushing as well. Results in [Crovella & Barford 1997] show that the automatic delivery of data changes the average traffic pattern on the network. This is rather obvious, prefetching and pushing will cause more traffic variability (sometimes called burstiness) since there will be periods of automatic transfers alternated with explicit requests due to faulty prediction. Between these periods there will be total passiveness when the user works with the data and the push mechanism has emasculated its predictions. The variability directly affects the average queuing sizes at the network switches, causing an increased packet delay. However [Crovella & Barford 1997] states that this negative side of prefetching can not only be avoided, it can be used to increase overall network performance compared with a situation where none such techniques are present. The way to do this is to use a transport rate limiting mechanism to control when to deliver the predicted data. If all prefetching and pushing takes place during the idle times between a finished transfer and a user request for more data the network traffic flow will be smoother and more evenly distributed over time. The result will be a network where the variability is kept low and the resources are better used.

3.3.2 Areas of Application

The usage of push technology today is primarily concentrated on web channels and automatic program distribution. The technology has been used in these areas with varying success. Below follows a discussion of the areas and how push has been used. There is also a short presentation of the usage of push in component distribution systems.

3.3.2.1 Pushing Web Pages

Automatic delivery of web pages was the original target of the push technology. At first it was used for web crawling and sending information via web channels. Later it also was incorporated in web marketing. More information regarding push technology on the web can be found in [Cerami 1998].

Web crawling essentially means that the user schedules automatic download of interesting web pages. The downloaded pages are stored locally and the user can browse them offline. In many cases the service includes support for differential updating of the locally stored information.

Web channels are a way to organise information on the web and automatically deliver it to subscribed users. First the channels were used solely for standard web pages but in time support for applets and applications where included in some solutions, this is discussed later in *3.3.2.2 Program Distribution*. The web channels are deceptive since they do not use true push, instead the client automatically asks the server if there is new information to download.

In web marketing push technology is used to send user and situation adapted advertising banners. This has been truly successful and is an excellent way to use push. The banner to send is chosen from simple models that usually are based on user information input by the user, what pages a user has visited and what services she has requested. After this the chosen banner is simply pushed to the client via the HyperText Transfer Protocol (HTTP).

The basic problem with the usage of push technology on the web is that it mostly consists of camouflaged pull sessions. That is, in most implementations there is no server automatically delivering data to the clients instead there is a list of scheduled pulls at the client-side. However in the identified area of web marketing true push is used in a powerful way. Another issue is the fact that the web represents a too big domain of information, making it difficult to predict what information the user wants to download. Though this is not entirely true due to people's preference to repetitive behaviour it still often causes a waste of bandwidth and presents unwanted information to the user.

What one can learn from the above is that a bad implementation of push combined with an unsuitable domain results in more harm than good. The bad reputation that push has is to some extent undeserved since the most well known implementations, i.e. web crawling and web channels, do not rely on true push.

3.3.2.2 Program Distribution

Automatic program distribution and updating probably is the most successful application of push technology. With such systems the administrator of a big organisation or company is permitted to avoid making individual installations, instead all of this can be handled automatically. This not only saves a lot of man-hours and money but also makes it possible to control what versions of the software the clients use.

One of the most famous solutions is Castanet from Marimba. It took push to a new level when it was released because it made it possible to deliver both software applications and information. Its main components are the Castanet Tuner and the Castanet Transmitter. The Castanet Tuner is the client software, it is free and included in Netscape's Netcaster. The Castanet Transmitter is a special server for broadcasting information to the tuners. To cut down the network traffic the system uses differential updates. Depending on how Castanet is configured the delivery is true server-initiated push or scheduled pull from the client-side. One way of looking at Castanet is seeing it as a fully automated Intranet. The necessary information can automatically be distributed to the employees together with updates of software applications. More information about Castanet can be found in [Cerami 1998] and on Marimba's homepage [Marimba 1999].

What one can learn from the area of program distribution is that a good implementation of push technology can be truly successful. The idea of using differential updating to compensate for the extra network traffic caused by erroneous pushing is appealing. Finally it is clear that there exists a market for systems with automatic versioning.

3.3.2.3 Component Distribution

The migration to server based component systems has opened a new market for push technology. In these systems the components must be distributed from the server to the clients. Preferably this would be done automatically, before the user's actions has led to an explicit request, otherwise waiting times will arise and the system will be perceived as slow and inefficient. Luckily the domain from where to choose components is limited. This indicates that push technology should be a good solution to integrate in such a system, provided that there is some way to make a good model of the users and that there exist some kind of order between accessed sets of components.

3.4 Caching

A common way to increase the efficiency of distributed systems is to use caches at different levels. In most systems there are two levels at which it is possible to use caching solutions, between the server and the clients and on the clients' local disks. The goal with a cache is to increase the local hit rate and decrease the network traffic. In this section some standard approaches to caching are presented, followed by more specialised solutions.

3.4.1 Standard Approaches to Caching

The success of a caching strategy depends mostly on how it replaces data when the cache is full and how fast its decision making is. Usually the replacement decisions are solely based on the current state of the cache. This is called using just-in-time information. Below follows a listing of some commonly used algorithms, see [Shaheen 1999] for more information.

- In the Least Recently Used (LRU) strategy the most recently used data is protected and objects that have not been used for the longest period of time are replaced.
- In the Least Frequently Used (LFU) strategy the least frequently used data is replaced first.
- In the First In First Out (FIFO) strategy a queue is used, resulting in that the data which arrived first also is replaced first.
- In the Last In First Out (LIFO) strategy the oldest data is protected and the newest is replaced first.

Today a lot of efforts are made to make caching algorithms that not only use just-in-time information but also utilise profile information. An example of such an algorithm is Least Relative Value (LRV). This is an extension of LRU, where every object is assigned a relative value. When the cache is full the data with the lowest value is replaced, thus it dynamically tries to evict what data to keep. When assigning values to objects in the cache, the algorithm weighs in many different factors. First the probability that the object will be requested again is estimated. Also the cost of bringing the object into the cache, based on size and time of transfer, and the cost of keeping the object in the cache, based on its size, are used. According to [Wong 1998] this algorithm has a good performance and easily beats LRU. The drawback with LRV is that it can be hard to implement.

3.4.2 Caching in a Distributed System

In a distributed system it is especially important to use caches. This could give considerably decreases in the perceived latency times.

There are some special issues that need to be considered when implementing caches in this type of environment. First of all it is important to identify at which levels to put caches. It is also important to evaluate the replacement strategy thoroughly since bad replacements will cause unnecessary network traffic and long waiting times. Another issue is consistency. If there are local copies of an object and an update occurs this could lead to inconsistencies.

In the following sections some different levels and types of caching are presented. These are all techniques that seem to be suitable to use in a system with component distribution.

3.4.2.1 Client-Side Cache

Caching at the clients is preferred since it gives good performance without adding to much complexity. The client-side cache should be placed in the local memory of the client if it is small enough. Otherwise the cache must be disk based. This results in slower cache access and requires the client software to have access to the local disk, which is not always true.

The usage of client-side caches is somewhat direct since all of the strategies in *3.4.1 Standard Approaches to Caching* can be used. In a component distribution system there are some specific aspects regarding client-side caching that must be considered:

- The cache needs to be integrated with the push operations. This includes avoiding unnecessary component transfers. Predicted components also need to be handled in a suitable way compared to requested and already used components.
- The size of the cache. If the cache is too small it will be ineffective. If the cache is too big it will require a lot of memory and thereby needs to be disk based. If the local disk is accessible, a big cache allowing the client to store all lightweight components will be very fast (once the cache is full). The cost of this is a high risk of network congestion when a lot of users log in at approximately the same time. The goal is to make the cache as small as possible while still getting the required performance.
- Should the cache be flushed when a user logs out? Local storage between the sessions could result in a performance gain in the beginning of each session. But it requires the cache to check its contents against the server to avoid using old, replaced versions of a component. Persistent caching easily introduces serious security flaws in a system and therefore generally is avoided.

3.4.2.2 Caching with Proxies

Today it is common to use proxies between the clients and an Internet connection, see [Luotonen & Altis 1994]. This saves bandwidth and cuts down the average transfer times.

A proxy essentially is a big cache that resides between the clients and the server. All requests from the clients are directed to the proxy, who if it does not have the wanted data downloads it from the server and before forwarding it to the client also saves it for future use. For this to work efficiently the proxy need to be able to serve as many requests as possible without having to use the server. The implications of this are that it must have a large cache and as many clients connected to it as possible.

If this idea were to be used in a server based component system it would probably not make any big difference in the perceived latency times at the clients. Though it would make the system more scalable since it lessens the burden of the server. In the best case scenario the proxy can store all components and handle all requests while the server takes care of prediction and pushing. The only time the proxy and the server need to communicate with each other is when a component is updated.

It should not be too difficult to implement a good proxy since this area is well known and commonly used in networks today.

3.4.2.3 Distributed Cache

An extension of traditional client-side caching would be to bind all local caches together into one big distributed cache. This needs some kind of message protocol handling the inter-communication. Also communication patterns need to be defined because it is not effective if all caches communicate with all others, causing a lot of extra load on the network. Probably these patterns will form some kind of hierarchy where caches at the higher levels keep tables over which objects are stored at the lower levels.

A big question is whether distributed caches should be limited to the client-side? It would probably be more efficient if several co-operating proxies where used. This makes it easier to find general communication patterns and limits the needed intelligence in the clients.

In [Engberg 1998] several distributed HTTP caches were evaluated. There were principally two solutions that performed well in the tests:

- Hierarchical cache. In this case there are a hierarchy of proxies and the clients send their requests to the nearest one. If the object does not reside at this location this proxy asks a proxy closer to the server if it has the wanted data. The request chain will eventually reach the server if the object is not found, which returns the data. On its way back to the originating source of the request the object is cached at the intermediate levels. Before a proxy returns an object to a client it checks with the server if the locally stored data matches its version, this ensures that the delivered data always is up to date.
- Summary cache. In this case each cache keeps a summary of the other participating caches' contents. This is done through multiple advanced hashing functions in an arrangement called Bloom filter. Using the summary a cache always knows which, if any, cache has the needed data. The updating is handled in a smart way where a cache that is to report a change of state only need to send its specification of the hashing functions and a declaration of which bits have changed to the all of the others.

A more detailed presentation of these strategies can be found in [Engberg 1998].

As can be seen a distributed cache is complex. It should only be used in large systems to increase the scalability. The advantages are shorter average transfer times and less pressure on the server. The price for this is extra communication over the network, though it tends to localise this traffic.

Distributed caches are hard to implement, requiring thorough analyses of possible communication patterns. These often result in ad hoc solutions needing initial configuration that one wants to avoid in a server based component system. It is also difficult to find security holes in systems using distributed caches, due to the introduced complexity.

4 PAX-E Observations

This chapter contains a summary of observations made during the analysis of PAX-E. Both the distribution model proposed and the implementation in this thesis are based on this knowledge.

4.1 Current PAX-E Implementation

Currently PAX-E uses fat clients and thereby does not transfer any workspaces from the server to the clients. However this will change in a near future, see *4.2 Future PAX-E Implementations*. Even if the current system needs modifications to be integrated with the distribution model in this thesis, it still can be used, as it is, to study how suitable such a combination is. For a successful integration PAX-E must be “user prediction”-friendly. This property is analysed in the following subsection.

4.1.1 User’s Behaviour in PAX-E

As mentioned in *1.3 The Target System* the users use different workspaces in PAX-E to solve tasks connected to workflow activities. One of the most important workspaces is the “to-do-list”, see *Figure 13*. This is an automatically updated, personalised workspace presenting a user’s incoming activities. Because of this the “to-do-list” is often used as a starting point, where the user checks out an activity. After this the user accesses the different workspaces necessary to solve the tasks associated with this activity. When an activity is finished the user usually returns to the “to-do-list” and repeats the procedure with the next activity. This cyclic behaviour combined with the fact that a user’s roles limit her work to certain types of activities, requiring certain workspaces to be accessed in a certain order indicates that a user should be possible to predict.

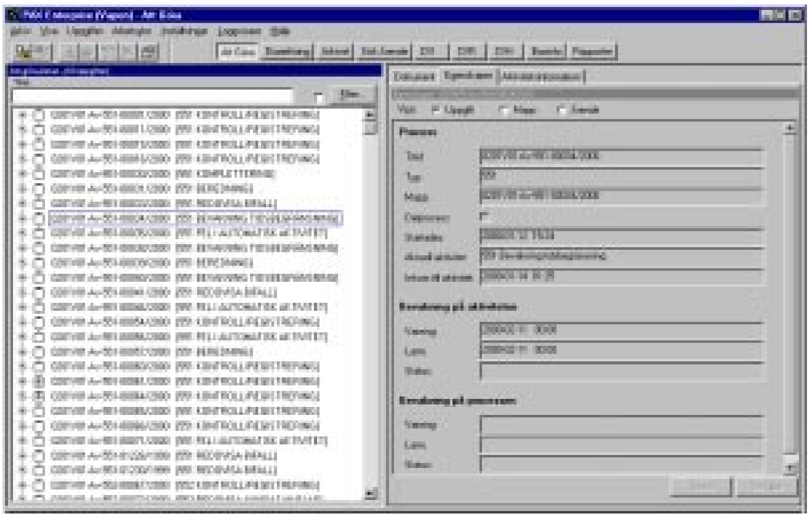


Figure 13. A “to-do-list” in PAX-E. The left part of the workspace contains a list of activities and the right part shows details about the selected activity.

Is it suitable to use user prediction at the workspace selection level of PAX-E? To answer this question the required properties mentioned in *3.2.1.4 Similarities Between the Areas* are checked. The result is:

1) The workspace selection in PAX-E is an easily identified atomic action.
2) The conclusion that users often use the workspaces in certain order was reached above. In other words, workspace selection patterns are very likely to occur in PAX-E.
3) There are only a few different workspaces in a PAX-E product. One can make the assumption that the distribution model does not need to handle more than ten different workspaces without limiting its usability in this platform. This does not mean that there cannot be more than ten workspaces in the system, but that one user will not need to have access to more than ten workspaces.

Everything indicates that PAX-E is a system highly suitable to utilise user prediction in. This user prediction should be used to predict the next workspace the user will select.

4.2 Future PAX-E Implementations

At the same time as this project has been done, Ida has been working with the next generation of PAX-E, denoted PAX Next Generation (PAX-NG). This will be a true server based component system implemented in Java. Although the system is still under development one can already approximate how many components each workspace will have and what sizes these will have. These are important observations, needed to make a realistic implementation.

The workspaces will require one to five components each. The size of the components will vary some, but the way they are combined should result in rather equal sizes of the workspaces, around 50 KB. This is a manageable amount of data to transfer automatically in the background while the user works uninterrupted.

It is likely that PAX-NG will provide various types of client software, since one solution will not fit all customers preferences. Examples on such solutions are clients implemented as Java applications and Java applets. PAX-NG also makes it possible to create personalised workspaces and workspaces tailor-made for a certain type of client platform.

5 Distribution Model

In this chapter the distribution model is presented. It consists of three parts: user profiles, push strategy and caching. All of these are discussed in detail.

5.1 Overview

Below follows a scenario that shows how the distribution model works for a user with an existing profile.

1) The user logs in to the system.
   a) The client sends login information to the server.
   b) The server authenticates the user and loads the corresponding user profile. The profile consists of owner information and workspace privileges. It also contains a number of subprofiles. Each of these represents dynamic data that is specific to a certain kind of client software. Depending on which platform the user currently uses, the corresponding data structure is used to configure the prediction mechanism.
2) The user selects the first workspace.
   a) The client sends a request of the necessary components. The dynamic data at the server is updated. The downloaded components are stored in the client's cache.
   b) When all components have been transferred the user can begin working with the workspace. Meanwhile the prediction mechanism uses the dynamic data to choose the workspace that the user is most likely to activate next. The corresponding components are automatically transferred to the client.
   c) The cache at the client stores the pushed components.
3) After a while the user selects a new workspace.
   a) The client checks if the necessary components already are available. If so the workspace is shown to the user. If the previous prediction was wrong or the push failed to transfer all components to the client, the necessary data will not be available in the cache and must be requested. After this the dynamic data at the server is updated.
   b) The workspace is shown to the user, meanwhile the server's prediction mechanism predicts another workspace and begins to push it.

Before the server initiates any component transfers it asks the client if any of the predicted data is locally stored. In this way unnecessary transfers are avoided.

In *Figure 14* different parts of the distribution model are illustrated. One can also see how they are interconnected.

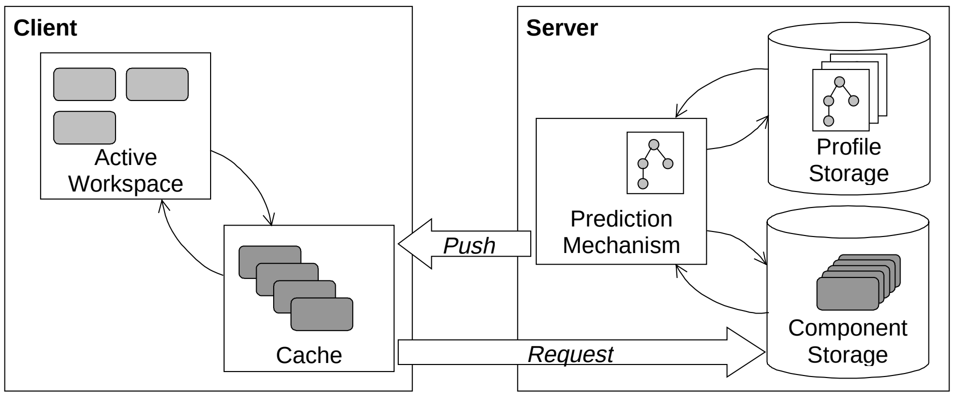


Figure 14. Illustration of the distribution model.

5.2 General Design Decisions

In *3.2.2 Design Issues* several design issues were mentioned that one should take into consideration when developing any kind of adaptive system. Below follows a short discussion around the different design decisions.

- The user model is individual. The reason for this is that in PAX-E users can belong to different groups and have many different roles, making it hard to make good group based user models.
- The user model is dynamic. Otherwise it will quickly become out-of-date when a user changes behaviour due to new working tasks etc.
- The user model is long-term since the collected knowledge is to be used in future sessions and not only in the current interaction.
- The user model collects knowledge implicitly. It is hard to create good questions that the user can answer in such a way that relevant information can be extracted. It is also desirable that as much as possible is handled automatically without causing inconveniences for the users and administrators.
- The user model is hidden. The collected knowledge about a user is difficult to present in a comprehensible way. This combined with the fact that the predictions are based on dynamic data that cannot be fine-tuned by the user explicitly makes the hidden model suitable.

5.3 User Profiles

The user profiles can contain three kinds of information about the user and her behaviour, as described in *2.2.1 Profile Information*: static data, dynamic data and session data. This section describes what the profile needs to contain.

The static data of the profile consists of the username associated with the profile and the corresponding user's workspace privileges. Since the handling of privileges can generate high administration costs it is important that it is well arranged. In systems like PAX-E it is efficient to store privilege information implicitly by defining which groups a user belong to in the profile. Mappings between groups and workspaces can then be kept separately in a database. Whether the workspace privileges are stored explicitly or implicitly is not relevant to the distribution model and one can choose the solution that suits the current situation best. The static part of the profile does not need to include which roles a user has in the workflow since the predictions are to be made on an individual basis, see *5.4.2.1 Initial Push Configuration*.

The dynamic data of the profile contains the user's history of workspace selections and statistics about her behaviour. This information is stored in a data structure suited for the chosen algorithm. The data structures used by the proposed algorithms consist of tables and trees. During one session a user is not able to change client software without logging out. This makes transitions between different types of tailor-made workspaces (only to be used on a specific kind of client platform) impossible. This observation, combined with the desire to keep the data structures as small as possible due to performance issues, implies that it is best to separate the dynamic data into subprofiles for each type of client platform. If one single, big data structure were used there would be a risk that it would grow too big to handle as the number of different platforms and workspaces increases during the lifetime of the system. This is prevented by the usage of subprofiles. When a user logs in, a new subprofile is created and initialised if the user has never used that kind of client software before. Otherwise the corresponding subprofile is loaded and used by the prediction mechanism.

The profile does not have to store session data explicitly since the different subprofiles represent the different types of client platforms a user has used. Any other kind of session-related information that might be interesting to use, i.e. network bandwidth, can be extracted during each session and does not need to be stored.

How the profile is to be structured is not practical to define in the model. A structure that works well in one implementation is not necessarily the best solution in another implementation. However it is probably best to store the subprofiles separately from the static part of the profile, making it possible to develop effective administration tools and to store the static part in a relational database. More information about the subprofiles and their contents can be found below.

5.4 Prediction and Push Strategy

The push strategy defines how the prediction mechanism will work. There are two questions that must be answered:

- How will the prediction be performed?
- When is prediction and pushing to be used?

These questions are analysed in the next subsections.

5.4.1 Prediction Algorithm

The choice of a prediction algorithm is crucial for the performance and requirements of the distribution model. It is hard to draw any conclusions about the performance of the different algorithms described in *3.2.3 Common Approaches* without testing them in this field of application. Therefore the final choice of prediction algorithm is postponed to *8.1 Algorithm Choice*, when both accuracy and speed have been evaluated in the testbench. The subsections below discuss the properties of the proposed algorithms, needed limitations and configurations.

5.4.1.1 Algorithm Properties

The strengths and weaknesses of the proposed algorithms are summarised in *Table 1*.

Table 1. Summary of the properties that the prediction algorithms have.

Algorithm	Strengths	Weaknesses
Markov	Very fast. Easy to implement.	Generally has bad accuracy since it only takes the current state into consideration when it makes a prediction. The data structure needs to be rebuilt if new workspaces are added.
IPAM	Prioritises new knowledge over older, maybe out-of-date, information.	The parameter α must be set explicitly. Only takes two previous workspaces into consideration when it makes a prediction. The data structure needs to be rebuilt if new workspaces are added.
LZ	Fast, provided that the tree has been updated.	Divides the history into blocks and therefore cannot identify all patterns. Occasionally needs to rebuild its tree, this is time-consuming and causes loss of gained knowledge.
PPM	Tries to adapt itself to the current situation by using a variable amount of history.	The orders of the different Markov models must be explicitly set. As the used orders increase the speed decreases and much more space is required. The data structures need to be rebuilt if new workspaces are added.
TDAG	Automatically identifies all previously seen patterns that match the current selection. The data structure does not need to be rebuilt when a new workspace is added.	Requires a lot of space. As its data structure grows its speed decreases.

Since the algorithm should learn incrementally LZ is inappropriate due to its restarting behaviour. In a server based component system this would result in widely varying performance. The rebuild is also a time-consuming operation that should be avoided. These are the reasons why LZ should not be used in the type of system targeted by this distribution model.

Two algorithms that have especially interesting features are TDAG and PPM. The ability to add workspaces without having to rebuild any data structures is a desirable feature of TDAG. The attempt of PPM to adapt its current predictions to the current level of accuracy makes it dynamic and might result in good performance.

5.4.1.2 Algorithm Limitations

In a PAX-E system there are many clients logged in at the same time. If the distribution model is to be used in this kind of environment it is important that the data structure of the chosen prediction algorithm does not require too much memory. Each active user needs a memory resident data structure to base predictions on. What is a reasonable limit for the memory requirement of an algorithm? This question is difficult to answer since the amount of available memory and the number of concurrent clients will vary a lot between the different systems. However discussions with personnel at Ida has resulted in an agreed memory limit at approximately 100 KB per each activated profile. This limit will be used as a guideline when the algorithms are analysed below. When reading these analyses, have in mind that there are at most ten different workspaces that the algorithm has to choose from when making predictions. This observation was made in *4.1 Current PAX-E Implementation*.

Markov does not have any problems with the memory limit. It will use a table with at most ten rows and ten columns when making predictions. In this data structure the frequency of occurrence will be stored as *integers*. An *integer* in Java requires 4 bytes. Thus the size of the data structure will be at most 400 bytes in a Java implementation.

IPAM uses the same data structure as Markov, although it stores floats in its table instead. A *float* in Java requires the same amount of memory as an *integer*. In other words IPAM will also need a data structure of at most 400 bytes in a Java implementation.

PPM extends Markov by combining Nth order Markov models of different dimensions. An Nth order Markov is easy to implement as a $N+1$ dimensional table of integers. If $N=4$ this would result in a data structure containing 100.000 elements, requiring 390 KB of memory in a Java implementation. This is more than the decided limit permits. But if first, second and third orders are used instead, the data structure of PPM will use 11.100 elements, requiring 43 KB instead. This is an acceptable amount of memory. PPM also needs a way to combine the separate Nth order Markov models. In *3.2.3.6*

Prediction by Partial Match two solutions were presented. Since the memory limitation will cause the algorithm to miss long patterns, due to the low orders used, the solution using one model at a time is best. Therefore PPM should use a preferred order, automatically increased or decreased depending on how well it performs.

TDAG differs from the other algorithms by using a tree and a vector instead. This tree needs to be limited in order to prevent it from growing infinitely. If a height limit of four is used, the tree contains at most 8.201 nodes ($1+10+10\cdot 9+10\cdot 9^2+10\cdot 9^3=8.201$). Each node consists of a workspace identifier and two counters. Nodes need to keep references to their children. An ordinary node implemented in Java will therefore require one *byte* as an identifier, two *integers* as counters and if it is not a leaf, nine memory references (32 bits each). Thus a node will require 45 bytes, if it is not a leaf in which case it requires only nine bytes. This results in a maximal tree size of 104 KB (there are at most 8.201 nodes and 7.290 of these are leaves). Besides the tree TDAG uses a vector to represent the current state. Height limitation is implemented by denying leafs to be added to the current state. This means that there are 911 nodes left. At most a tenth of these can contain the current workspace identifier and thus belong to the current state. Therefore the current state can maximally contain 92 memory references, requiring 368 bytes of memory. The conclusion is that a TDAG with a height limit of four will at most require 105 KB. This is an abnormal situation very unlikely to occur, hence TDAG can be allowed to use a tree of depth four in this distribution model.

All algorithms are to be prevented from making “strange” predictions. In the case with TDAG this means that no predictions based on nodes with an `out-count`, the counter telling how many times the node has been placed in the current state, equal to one will be made. Without this threshold, predictions based on seldom visited nodes could seem better than predictions based on frequently visited nodes. The other algorithms also need a similar kind of limitation, preventing them from making predictions when all the considered alternatives has a probability equal to zero. Otherwise predictions based on the order of the workspace identifiers in the tables will be made in the early stages.

5.4.2 Push Policy

Is it suitable to push a new workspace before the currently predicted workspace has been used? Since this prediction will be based on a prediction, which has not been proven to be correct so far, a lot of uncertainty is introduced. Bad predictions do not shorten the perceived latency and should be avoided as much as possible. In this way unnecessary network traffic is avoided. The conclusion is that a new prediction should not be made until the result of the current prediction is known.

The usage of thresholds to identify situations where predictions are good enough to be realised by pushing is a good way of limiting waste of network bandwidth. However such thresholds are very difficult to define and they require rigorous testing. Beside the thresholds mentioned earlier in *5.4.1.2 Algorithm Limitations*, the proposed prediction algorithms are not designed for such usage either. Therefore pushing should always be performed when a prediction has been made.

The push operations should start immediately after a prediction is completed. Since the predictions are made on a workspace selection basis, it is best to update the dynamic data of the profile as soon as the server has been informed of a user's workspace selection. This is followed by a prediction based on the current situation and finally the chosen workspace is pushed.

If a push fails to finish before the client requests the next workspace it is to be stopped immediately and replaced by the requested component transfer. The reason for this is that the corresponding prediction is old and does not apply to the new situation.

5.4.2.1 Initial Push Configuration

The initial push configuration has not been covered by the push policy so far. This is a special case, defining how a new profile or a new subprofile is to be used in the system. There are several solutions to how pushing should be handled when a user's behaviour is not known yet. The alternatives are:

- Push based on groups and learning phases. This solution is based on the fact that similar users often can form groups. These groups are used together with learning phases of a suitable duration. For each group the access pattern of the first user during the learning phase is stored. After this it is used to initialise the dynamic data of all new users that belongs to that group.
- Push based on frequency. The pushing is based on the frequency of usage of the different workspaces; meanwhile the dynamic data is updated. When enough knowledge has been collected the normal prediction mechanism takes over.
- Role based push. In this solution there are initial dynamic data corresponding to the different roles in a workflow system. Every new user profile is initialised with dynamic data typical for the behaviour associated with her roles.
- No push in the initial phase. The pushing is not started until the algorithm has collected a sufficient amount of data about the user's behaviour and is positive about making the right prediction.

If push is based on groups and the first user of every group is used to build the typical profile for that group a number of problems occur. One problem is how to be sure that the first user uses the system in the intended way. Another problem is that in a workflow system a user can have many kinds of roles. This implies that it is hard to create user groups where all members work with the system in the same way. Therefore the usage of push based on groups is not suitable in PAX-E.

Push based on frequency is easy to implement. A threshold value can be set for the number of times, after which sufficient data about the user has been collected. This approach does not use workspace selection patterns in the initial phase causing the predictions to be uncertain. As mentioned in *4.1.1 User's Behaviour in PAX-E* PAX-E users often work in cyclic workspace selection, making push based on frequency unsuitable.

If the push is based on roles, some initial profiles are needed for each role. An advantage with this approach is that the administrator is given more control over the initial profiles, compared to the solution with a learning phase. However the administration costs are high since these initial profiles have to be manually configured. The roles in PAX-E are strongly connected to the customer's organisation, requiring new initial profiles to be created for each installation. Since one of the project requirements is to avoid causing more administration work, role-based initial pushing is not suitable to use in this model.

One way of minimising the administration costs is to disable the push operations until a sufficient amount of data has been collected about the user. Even if typical workspace selection patterns for roles and/or groups can be identified, the profiles are built purely on the user's own actions and not on an expected behavioural pattern. The only drawback with this approach is that users can experience waiting times during the initialisation. However provided that it is possible to correctly define how long the push should be disabled and that the users are well informed on how the system works, this solution provides automatic individual customisation suitable for PAX-E. An easy way to decide when it is time to activate the pushing is to count the number of updates made to the prediction mechanism's data structure and compare it to a threshold value. Which threshold value to use cannot be decided until a prediction algorithm has been chosen.

5.5 Caching

The caching policy consists of a client-side cache that supports the pushing. Of the cache strategies mentioned in *3.4.1 Standard Approaches to Caching* the LRU solutions seems promising. The reason for this is that the way a user works in PAX-E tends to make him/her access workspaces that have been recently used. FIFO also seems suitable, due to the way pushing works. In ordinary push systems newly pushed data always replace older predicted data. The final contender is LRV, normally outperforming LRU. But in this field of application this solution is not fitting, requiring the client to predict how likely a re-transfer of a component is. Predictions should only be made at either side of the system in order to avoid conflicts.

Distributed client-side caching, mentioned in *3.4.2.3 Distributed Cache*, is complex and almost always need to be tailor-made for each system. Therefore it does not seem to be general enough to include in this type of model. Also it increases the network load, which probably should be avoided in a server based component system that already uses a considerable amount of network bandwidth.

If the LRU and the FIFO strategy could be combined and adapted to the usage of push, this should result in a good cache solution at the client. It is desirable to replace earlier false predictions and components that have not been used recently. The easiest way to combine the strategies and to accomplish this property is to divide the cache into two separate layers. The layers can then be used to separate predicted components that have not been used yet from the ones that actually has proven themselves useful. The parts of the cache are denoted: prediction layer and active layer.

The prediction layer is essentially a FIFO cache with a queue ordered by the time of arrival. When a new prediction is made the cache checks if the components already are available. If a predicted component exists in the cache, there is no need for re-sending it from the server instead it only needs to be repositioned in the queue. On the other hand, if a predicted component is not in the cache it is sent and placed in the queue of the prediction layer. If there is not enough space to store the component, the oldest component is removed since it is either part of a false prediction or has not been used for a long time.

The active layer stores components that are currently used or have been used at least once. The basic principle for its operation is LRU. When the client makes a request for a component, there are the following three possibilities of where to place the component:

- If the component already is in the prediction layer, it changes place with the least recently used component in the active layer. This component is placed in the queue and thus recycled instead of deleted. In this way network bandwidth is saved if it is part of a near future prediction.
- If the component already is in the active layer, the priority in that layer is updated so that the requested component is marked as most recently used.
- If the component is not in the cache it needs to be sent from the server. It is then placed in the active layer and marked as most recently used. If there is not enough space in the active layer, the component marked as least recently used is replaced and moved to the prediction layer.

Figure 15 shows the structure of the 2-layer cache.

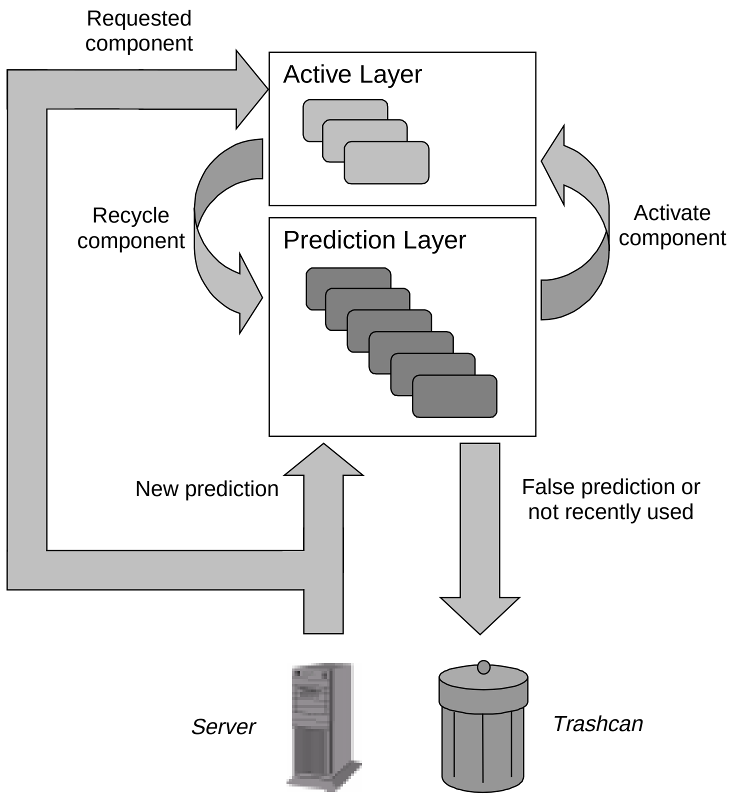


Figure 15. The handling of components in the 2-layer cache.

The design of this 2-layer cache makes it possible to avoid unwanted replacement of the most recent predictions. By using a pure LRU cache this can be a problem, since in this strategy the order of the components is based on the usage and the predicted components are not always used.

Due to reasons mentioned in *3.4.2.1 Client-Side Cache* the size of the cache should be as small as possible. The active layer must be able to store a complete workspace in order to work as described above. The prediction layer must be able to store more than a complete workspace otherwise the recycling of components will be disabled. This indicates that a suitable cache configuration would be to have an active layer with the size of the biggest workspace and a prediction layer twice as big. This size should be limited, permitting the cache to be memory based. The cache is always flushed in the end of a session. This avoids the requirement of disk access and the introduction of security flaws.

Large systems with a lot of clients will probably need several load balanced servers. And in some cases component proxies, see *3.4.2.2 Caching with Proxies*, might be needed to handle explicit component requests and thus lessen the burden on the servers. This kind of caching needs to be created individually for each system and will therefore not be included in this distribution model.

6 Implementation

This chapter presents the implementation made in this project. The language used was, as mentioned earlier, Java. The purpose of the implementation was to evaluate the distribution model presented in this thesis and to be able to choose the most suitable prediction algorithm.

First a system overview gives both a functional and a technical presentation of the implementation. The latter one shows the different parts of the system and how these work together. It contains a user case, how the communication between client and server is done and detailed information about each part. A section about logging follows this showing what is logged and how necessary statistics can be extracted from the log files. Finally some limitations of the implementation are presented.

6.1 System Overview

The implementation is a distributed library system. However it is designed for usage at a computer company, like Ida, where it is not only interesting to store books and magazines in the system but also software and electronic publications. It is also easy to configure the system for more traditional usage at a public library etc. The implementation is a multi-user system, although at the present time only one user at a time can login.

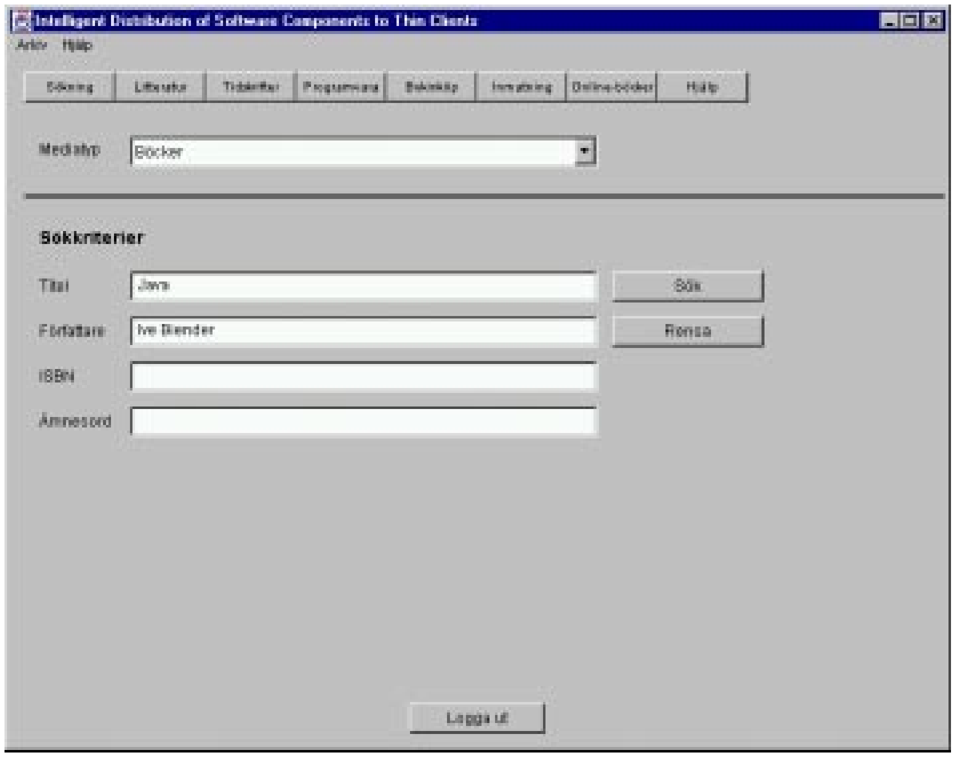
6.1.1 Functionality

The functionality is, as in PAX-E, divided into different workspaces. Each of these contains components that together provide all functionality needed by the user to solve one task. There are eight different workspaces making it possible for the user to:

- Perform media dependent or independent searches among available books, magazines and software. It is also possible to get detailed information about specific search hits.
- List comprehensive information about all available books.
- List comprehensive information about all available magazines.
- List comprehensive information about all available software.
- Order a new book.
- Store a new item (book, magazine or software).
- Read electronic publications.
- Get help.

The user can choose workspaces from a selection toolbar in the main window. The currently active workspace is also shown in the same window. There are also separate windows presenting search results, detailed information about a search hit and for logging in. For testing purposes it is also possible to view the client’s cache contents and its actions in a concurrent window.

The implementation is a multi-user system where workspace privileges can be set on an individual level. Thus it is possible to completely hide functionality for users that should not be aware or allowed to perform certain operations. *Figure 16* shows a screenshot from the implementation.



Mediatyp: Böcker

Sökkriterier

Titel: Java
Författare: Ive Bender
ISBN:
Ämnesord:

Figure 16. Screenshot of the media search workspace. The toolbar at the top allows the user to select between the available workspaces.

6.2 Technical Architecture

In *Figure 17* the technical architecture of the implementation is illustrated.

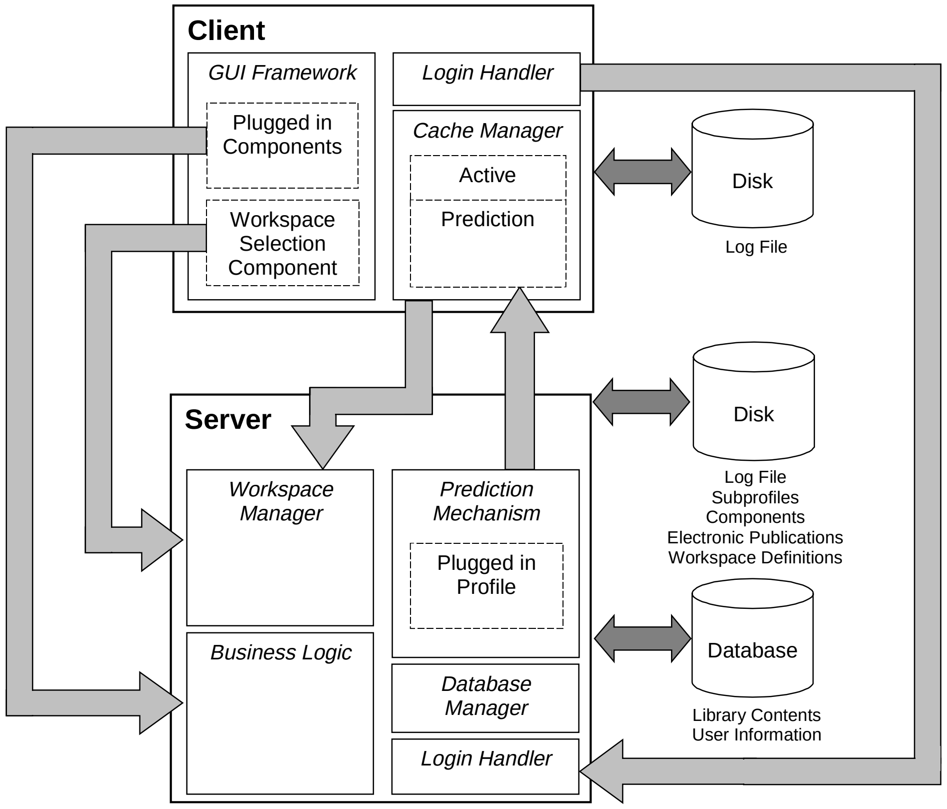


Figure 17. The main parts of the client and the server in the implementation. The arrows show remote method invocations.

Section *6.2.1 User Case* describes how the different parts of the system work together. Then sections describing each part of the system in detail follow.

6.2.1 User Case

When the server is started it configures itself by using a workspace definition file. After this the following scenario takes place:

1) The login handler at the client asks the user for username and password. When these have been entered the corresponding handler at the server is invoked.
2) The login handler at the server uses the database manager to validate the information entered by the user. If the login is invalid the procedure restarts, otherwise the stored information about the user is used to plug in the user's profile into the prediction mechanism. The user's privileges are also used to initialise the same mechanism and to configure the Workspace Selection Component (WSC). In this way the user will not be aware of any other workspaces than those she may use.

3) The client triggers the cache manager to get the WSC from the server. This is done by a message to the workspace manager at the server. It returns the personalised WSC.
4) The WSC is plugged into the Graphical User Interface (GUI) framework and activated. Now the user is presented with a toolbar in which workspace selections can be made.
5) When the user selects a workspace the WSC asks the server's workspace manager which components the client needs. A list is returned and passed to the cache manager.
6) The cache manager checks if the components are available. Since no predictions have been made yet, the cache is empty and the manager therefore asks the server for all of the needed components.
7) The cache manager places the returned components in the active cache and tells the GUI framework that there are new components to plug in.
8) The GUI framework activates and shows the components, thus the selected workspace is now available for usage. Feedback is sent to the prediction mechanism, this is used to update the user's profile with the workspace selection. At each time the mechanism receives feedback it predicts which workspace the user is most likely to select next. The components belonging to that workspace are pushed, if necessary, to the prediction cache of the client. When receiving pushed components the cache manager removes old contents from the prediction cache if this is necessary.
9) When the user uses the components, they invoke the business logic of the server to get results of needed calculations and database queries. Upon return these are shown in the workspace.
10) When the user selects another workspace the procedure is repeated from step five. However since a prediction has been made the client may not have to request any components in step six.

If the cache manager requests a component explicitly at the same time as the server is performing a push operation, the push is aborted and replaced by pull.

6.2.2 Client/Server Communication

The implementation uses RMI to handle all communication between the client and the server. The reasons for this are:

- The application is not a large distributed system; only one client at a time can log in on the server. Therefore the system will not suffer from performance degradation due to RMI being unsuitable to serve many clients at the same time.
- The objects passed between the server and the client mostly have a small size, with exceptions for a few of the biggest components. Therefore the performance difference between RMI and CORBA should be small.
- Everything is implemented in Java. RMI is by far the easiest way to make a distributed application in this language.
- The development time was too short to permit the more advanced approach using CORBA.

6.2.3 Data Storage

There are two types of storage in the system. Some of the data is well suited for storage in a database, while other data is more practical to store directly in the file system of the server.

6.2.3.1 Database

The user accounts and all library data (information about books, magazines etc.) are stored in an Oracle Lite 3.5 database. The user account consists of a user's username, password and the static data in the profile. For more information about the profile see *6.2.5 Profiles*. The part of the server that handles this database is called the database manager. It uses the built-in Java DataBase Connectivity (JDBC) package to connect to the database and make queries.

6.2.3.2 File System

The data that is stored in the file system is divided into two different types of files, ordinary text files and serialised Java files. The text files are used to:

- Log results. This is done at both the server and the client. More information about logging can be found in *6.3 Logging*.
- Define the different workspaces. A workspace definition contains the name of the workspace, a unique identifier, a list of what components it contains and a path to where those components can be found.
- Store electronic publications. In the current version a text file based format is used for storing electronic publications. An electronic publication consists of several files: a table of content file and individual chapter files containing the bodies of each chapter.

Serialised Java files are used to store the components of the workspaces and the users' subprofiles. Serialisation is a built-in Java feature that permits the programmer to store the current state of an object. After an object has been serialised to file it can be restored and used in any Java program at any time. However the program must be aware of the corresponding class definition to be able to interpret the serialised file correctly.

6.2.4 Components

The components in the workspaces are implemented as Java Beans.

> *"A Java Bean is a reusable software component that can be manipulated visually in a builder tool."*
> *[Horstmann & Cornell 1998 p. 333]*

In the implementation two types of Java Beans are used. The first type of component is one or a combination of several user interface components. These are interconnected in the Java Bean to work together and appear as one component. The second type of component is called dummy component. These are invisible components that do not provide any kind of functionality. They are put into some of the workspaces to make the implementation as similar to the future version of PAX-E, mentioned in *4.2 Future PAX-E Implementations*, as possible in terms of the number of components and the size of them. One to five Java Beans are combined in each workspace, resulting in workspace sizes between 30 and 60 KB.

6.2.4.1 Infobus

In most workspaces several Java Beans need to communicate with each other and/or the GUI framework to be able to provide the needed functionality. This is achieved with an Infobus, a solution that makes it possible to interconnect Java Beans and dynamically exchange data between them. The protocols used are based on the notion of an Infobus, hence the name. Infobus is free to use and currently distributed by Sun Microsystems at [Sun 1999/3]. In *Figure 18* the components of a workspace and their communication ways are illustrated.

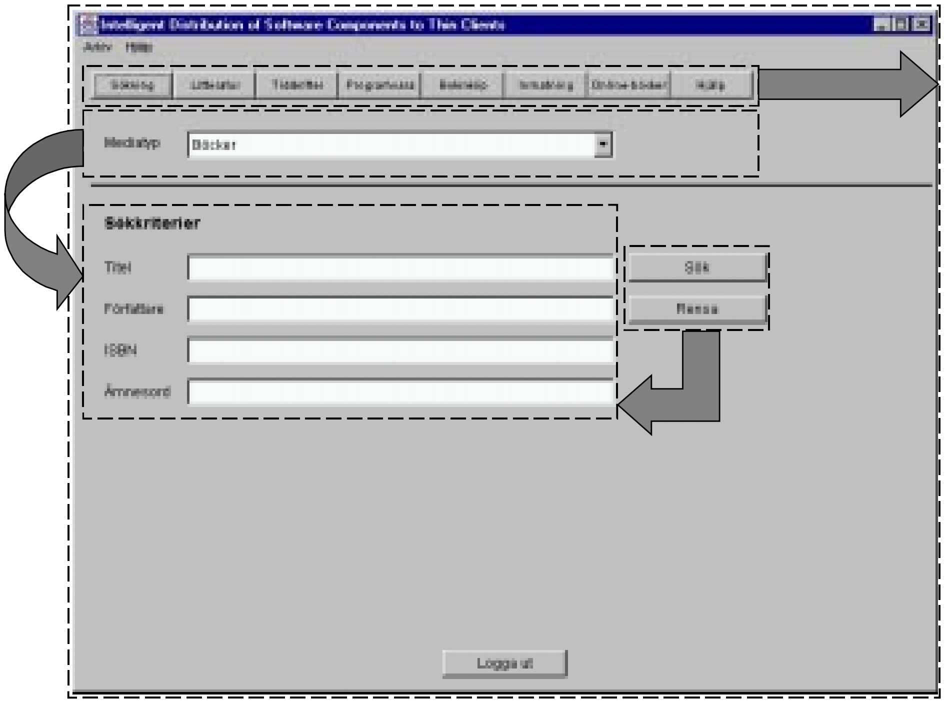


Figure 18. The dashed lines mark the different components. Arrows symbolises Infobus communication.

6.2.5 Profiles

In the implementation the profiles are split into two separate parts. One part contains the static data and is, as mentioned earlier, stored together with the user account in the database. The other part consists of an object containing a prediction algorithm and the dynamically updated data structure it uses to model the corresponding user. This object corresponds to the subprofile in the distribution model. Since there is only one type of client software involved, each user has one subprofile. The subprofile can predict the user's next workspace selection, update its data structure according to the result of the last prediction and prepare itself for storage. To minimise its size the subprofile uses small unique workspace identifiers instead of full names.

As mentioned in *6.2.3.2 File System* the subprofiles are stored using serialisation. This makes it particularly easy to use different solutions for making predictions in the system. Since everything that is specific for one solution is encapsulated in its specific profile one does not need to change anything in the prediction mechanism to try another solution in the future. The drawback of this approach is a slight overhead due to the fact that the same algorithm definition might be stored in several different subprofiles. However since almost the subprofiles' entire file size is caused by the data structure that is different in all profiles, this is not as serious as it might seem.

The subprofiles do not return the identifier of the workspace that seems most probable in the current situation. Instead a probability distribution is returned. It contains the probabilities for all workspaces that currently exist in the system. The reason for this is that otherwise a profile needs to be aware of the corresponding user's workspace privileges. This would make it more difficult to change privileges and it would also slow down the predictions.

6.2.6 Business Logic

All business logic in the system is collected at the server. It is divided in managers responsible for different types of calculations. When the user uses a component in a workspace in such a way that business logic is required, the component invokes the correct manager at the server and the result is computed and prepared for visualisation. Then it is returned to the component, which presents it to the user.

6.2.7 Prediction Mechanism

The prediction mechanism is responsible for handling the subprofiles. It loads the subprofile data from disk and plugs it in when a user logs on. During the logout sequence the subprofile is serialised to disk to save the current state until next login.

When the mechanism is not loading or storing subprofiles it is always waiting for feedback from the client. The feedback consists of the identifier of the currently active workspace. When such feedback is received the server starts a concurrent thread dedicated to the prediction mechanism. It forwards the feedback to the currently plugged in subprofile. This triggers the subprofile to update itself. After this the mechanism asks the subprofile for a probability distribution based on the current situation, see *6.2.5 Profiles*. The current user's privileges are used to select the most probable (and allowed) workspace from the distribution. The mechanism now asks the workspace manager about which components that belong to the predicted workspace. Before any components are pushed the mechanism asks the cache manager if any of them already are stored at the client. Finally the components that really need to be transferred are pushed. After this the prediction thread is disposed. Component requests from the client interrupt and automatically dispose prediction threads, if such are still running. The reason for this is that the push handled by such a thread is either transferring the wrong components (due to an erroneous prediction) or has chosen the correct components but initiated the delivery too late.

6.2.8 Cache Manager

The cache manager provides the client with the 2-layer cache described in *5.5 Caching*. If a needed component is not available in the cache, the manager requests it from the workspace manager at the server.

The active layer requires 61 KB memory and the prediction layer 122 KB. These sizes where determined by the size of the components in the largest workspace.

6.2.9 Graphical User Interface Framework

The GUI framework at the client contains the different windows of the client; see *6.1.1 Functionality* for information about which these are. The most important of these is the main window where the user works with the current workspace. Before the user can use a selected workspace the framework needs to activate that workspace. The activation procedure starts by a deactivation (resets the communication provided by Infobus) and removal of the current workspace's components from the main window. Thereafter the new components are loaded and activated (configures the communication provided by Infobus). The activation of some components also includes initialisation utilising the business logic at the server. After this the workspace is plugged into the framework, now it can be shown to and used by the user.

For the workspace selections to work the WSC and the GUI framework need to co-operate. This is solved by an Infobus connection between them. This connection is configured during the activation of the WSC and exists during the whole session.

6.3 Logging

The performance of the implementation is measured in several different ways at both the client and the server. During a session intermediary timing results and information about the predictions are written into log files. At the end of each session extracted statistics are appended to the logs.

6.3.1 Client-side logging

During the session the following things are measured and logged for each workspace selection:

- Workspace name.
- The waiting time measured from the moment the user selects the workspace to when it is possible to start working with it. This includes the time it takes to deactivate and remove the former workspace, the time it takes to load and/or transfer all components to the client and the time it takes to activate and show the workspace.
- Time to deactivate the former workspace.
- Time to activate the workspace.
- The number of components that were available in the client's cache.
- The number of components that the client had to request from the server.

At the end of each session a summarisation is written containing the following:

- Total number of components that existed in the cache when needed.
- Total number of components that had to be requested from the server.
- The cache hit ratio.
- Total waiting time.
- Total time spent activating and deactivating workspaces.

The hit ratio of the cache is calculated by dividing the number of components that existed in the cache when needed by the total number of used components. In this case it not only measures the efficiency of the cache since it depends heavily on the prediction quality.

6.3.2 Server-side logging

During the session the following things are measured and logged for each workspace selection:

- The result of the last prediction.
- The time it takes to update the profile.
- The identifier of the predicted workspace.
- The time spent on decision-making.
- The number of pushed components versus the total number of components in the predicted workspace.
- The time it took to push the components.

At the end of each session a summarisation is written containing the prediction ratio. That is the number of successful predictions divided by the total number of workspace selections. This measures the efficiency of the prediction mechanism.

6.4 Limitations

There are some limitations to the implementation that one should be aware of, these are:

- Concurrent Java threads perform all timing. This introduces time-sharing that results in uncertain times. The logged times can therefore not be solely trusted, but they are good enough for comparisons between different solutions in the same implementation.
- The system would probably be marginally more efficient if CORBA were used instead of RMI.
- It is possible that a socket solution instead of object passing through RMI can reduce the time it takes to transfer components.
- At the present time the server does not allow more than one session at a time. This makes it impossible to test such things as scalability etc.

7 Evaluation

Running the implementation with sequences of workspace selections has been the basis for an extensive evaluation of the algorithms and the cache. The workspace selections simulate a user working in the system as she moves between the workspaces to perform tasks. This chapter presents the objectives, the test cases used and the results.

7.1 Objectives

The main objective is to identify the best solution based on the distribution model. To be able to do this several test cases and configurations have been designed. Together these should reveal characteristics of the algorithms and the cache making it easier to weigh advantages against disadvantages. The tests also show the level of performance achievable by combining the client-side cache with the prediction mechanism at the server.

7.2 Measurements

The performance of the algorithms is measured by the prediction ratio that is defined in *6.3.2 Server-side logging* as the number of correct predictions divided by the total number of predictions. This ratio is recorded in the server-side log. However one must also take into consideration the size of the corresponding profile and the time it takes to update the profile and make predictions.

The performance of the cache is measured by its hit ratio which is defined in *6.3.1 Client-side logging* as the number of times when a needed component is found in the cache divided by the total number of times a component is needed. This ratio is recorded in the client-side log. Clearly the measured hit ratios are dependent on how well the prediction mechanism performs, it is therefore important that all comparisons are made using the same push situation. At first thought, one could easily think that the cache performance should be measured without using any server initiated component transfers, but this would not give a fair picture since the cache has not been designed for this type of situation.

The waiting times at the client are the best way of measuring the overall performance of the total solution. In this implementation such waiting times can be extracted by using the client-side logs. The waiting time is calculated as the difference between the total waiting time and the time spent activating and deactivating workspaces. The reason for this subtraction is that the latter time is a constant overhead that is not influenced by different combinations of the individual solutions.

7.3 Tests

The test cases have been run several times using different test configurations. Reference sessions, where both the cache and the prediction mechanism are disabled, have also been run in each case. Below follow descriptions of the different test cases and test configurations used.

7.3.1 Test Cases

The test cases consist of both artificial tests and real world simulations. All tests are divided into two sessions. The first session is used to train the profile. The second session is used to test the profile. It is the latter session from which the measurements are taken.

The artificial cases are constructed to test certain properties of the different algorithms. These are completely fictitious sequences of workspace selections without any connection to the functionality of the implementation.

To find out how the algorithms perform in a real system, two kinds of simulations have been constructed. The first simulation consists of workspace selection patterns likely to occur if the implementation is used as a library system. The second simulation is based on an analysis of one of Ida's PAX-E solutions.

Some of the test cases include white gaussian noise, which is a randomised sequence of workspace selections. This is used to model a user clicking around planlessly among the workspaces. The randomisation is made from an evenly distributed probability. The term noise will be used from now when referring to such randomised sequences.

In the examples below sequences of workspace selections are represented by combinations of workspace identifiers, i.e. *AB* represents the situation where a user starts working in workspace *A* and then continues in workspace *B*. A plus sign (+) marks the end of an initialisation and the start of the test session. For readability comma signs (,) have been inserted to mark each pattern.

7.3.1.1 Artificial

The artificial test cases are:

1) Adaptability to pattern change. In these tests the profile is initialised with a sequence containing a repeated pattern. After this follows a test session in which the original pattern has been replaced by a similar one. The prediction ratio in the tests shows how well the algorithm adapts when a user changes behaviour. The tests use different patterns, type of transformations and sequence lengths. A small example of this kind of test is: *ABCDEF,ABCDEF+BACHEF,BACHEF.*
2) Noise sensitivity. These tests consist of two parts. In the first part the profile is initialised with a sequence that starts with the repetition of a pattern and ends with noise. After this follows a test session in which only the regular pattern occur. The second part extends the test and makes it more realistic by using sequences containing several different patterns, randomly ordered and repeated. In this part the training sequence also ends with noise (that does not exist in the test session). The prediction ratio in

the tests shows how sensitive the algorithm is against interruptions in a user’s normal behaviour. This situation could occur in PAX-E when new users start to use the system and/or if a user needs to perform small duties beside the normal tasks. Different patterns and length of the noise have been used in these tests. A small example of the first part of the test is: *AGFE,AGFE,CBGBFEHA+AGFE,AGFE*. A small example of the second part of the test using the patterns *H*, *DEG* and *BCDE* is: *H,DEG,DEG,H,BCDE,DEG,BCDE,H,BDFACABHGFEGBCDFHD+ BCDE,H,BCDE,DEG,H,DEG,DEG,H*.

3) Handling of irrationality. In these tests the profile is trained and tested with a long sequence of noise. The prediction ratio shows how the algorithm handles total randomness. Several noise sequences of different lengths were used in these tests.
4) Long patterns reminding of each other. These tests use sequences of varying length that contains a mix of long, similar patterns. The length of some patterns is intentionally chosen so that the algorithms cannot classify the whole pattern as one unit. The prediction ratio shows how good the algorithm is at identifying and separating patterns. A small example of this type of test using the patterns *ABCDEHA*, *ABDEFHG* and *AHCD* is: *ABCDEHA,ABCDEHA,ABDEFHG,AHCD,ABDEFHG,AHCD,AHCD+ ABDEFHG,AHCD,ABCDEHA,ABDEFHG,ABCDEHA,ABCDEHA,AHCD*.

Each test case has been used to generate test suits with up to ten individual tests.

7.3.1.2 Simulations

The simulations show how well the different algorithms would perform in real systems of the targeted type. They have been based on analyses on how users are likely to behave in the implemented library system and a PAX-E system that recently has been delivered by Ida to Statens Kriminaltekniska Laboratorium (SKL). The results of the SKL analyses are user model charts that can be used to generate long, realistic working sequences.

The library system simulation consists of eight different patterns of lengths varying from one to four, although most of the patterns have a length of three or four. Therefore this test probably requires an algorithm that handles long patterns well otherwise the profile can have problems separating the patterns of equal length from each other. The patterns are randomly combined into two different sequences. One used as a training session containing 500 workspace selections and a second used as test of length 1,000.

The SKL simulation is more realistic and advanced. It is based on three subtests. Each subtest is connected to a user with one or two roles at SKL. The workspaces used at SKL are mapped to the workspaces of the implementation. The functional differences between these two systems does not affect the tests since it is only interesting to simulate the behaviour of a user in terms of movement between eight workspaces. The first subtest is based on likely behaviour of an administrator of the system. It contains patterns for solving typical administration tasks. However it is very common that an administrator sometimes works as a registrar. Therefore this subtest also includes the possibility of role changes and thereby it also shifts between two sets of patterns. The second subtest is based on behaviour likely to occur for a dedicated registrar. It contains patterns for solving typical registrar tasks. The third subtest is based on the behaviour of a regular user.

Figure 19 shows the chart used to generate sequences for the first subtest.

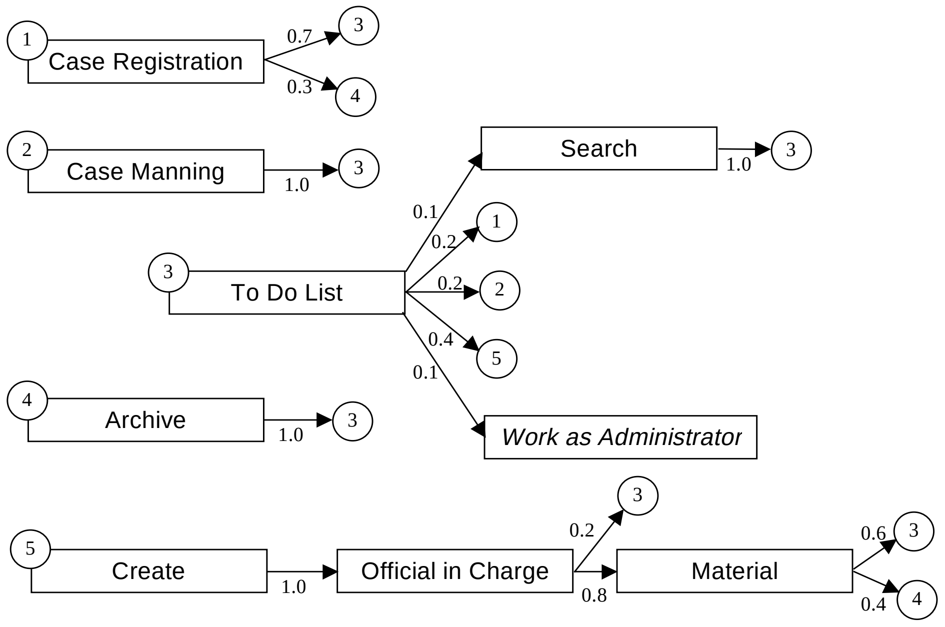


Figure 19. User model chart for a typical administrator at SKL. The boxes represent workspaces. Arrows symbolises likely transitions between these. Each transition is associated with a probability of it being realised. The box named “Work as Administrator” symbolises a transition to a chart for administrators.

The subtests use two sequences each, a training session with 500 workspace selections and a test session of length 1,000.

7.3.2 Test Configurations

This section describes the different system configurations that have been used during the evaluation.

7.3.2.1 Algorithm Configuration

To be able to evaluate how the chosen algorithm limitations in *5.4.1.2 Algorithm Limitations* affect the performance of TDAG and PPM the simulation tests also were run with other algorithm configurations. In the TDAG case the performance of the chosen TDAG profile, using tree depth four, was compared with an extended profile, using tree depth six. In the PPM case the performance of the chosen PPM profile (using first, second and third ordered Markov) was compared with an extended profile (using first, third and fifth ordered Markov). The extended profiles thus have the capability of identifying more and longer patterns.

To be able to evaluate how the performance of IPAM is affected by the value of the parameter α the simulations where run with profiles using different values. Recall that the value of α tells how much IPAM’s predictions depend on earlier collected data about the user, see *3.2.3.4 Incremental Probabilistic Action Modelling*. The values used are 0.7, 0.8 and 0.9. The reason for this is that 0.8 proved to be the optimal value in [Davison & Hirsh 1998]. Even if the optimal value varies between different fields of application it should be close to 0.8 in this implementation too.

These tests where executed with an active cache and a prediction cache of equal size (with enough space to store the largest workspace in each).

7.3.2.2 Cache Configuration

The simulation tests were used to measure the performance of the client-side cache presented in *5.5 Caching*. This was done to validate that the cache performance was adequate, using an active layer of the same size as the largest workspace (61 KB) and a prediction layer twice as big.

7.4 Results

The results from the different tests are presented and analysed in the following subsections.

7.4.1 Algorithm Configurations

This section answers the question if the chosen algorithm configurations are reasonable in terms of performance. All of the simulation tests were used in this evaluation.

7.4.1.1 IPAM

The results presented below in *Table 2* show that the general guideline for the value of α seems to hold in this field of application. The performance could probably be increased slightly if the value of the parameter was to be thoroughly appointed with a higher precision. This is only worth the effort if this algorithm is chosen as the best solution and will therefore be postponed. In the rest of the tests α is set to 0.8.

Table 2. Prediction ratio averages from all of the simulation tests, using different configurations of IPAM.

α	**Prediction Ratio**
0.7	47.3%
0.8	49.4%
0.9	47.3%

7.4.1.2 PPM

The size of a profile using the standard PPM configuration is approximately 40 KB. The extended profile increases this to almost 5 MB. This is too big to handle in a multi-user system where many clients are permitted to log in at the same time. It is also worth to mention that the large profile slows down the log in and out procedures noticeably, though the time it takes to update and make predictions is not increased to an unacceptable level. Does the extra space give an equally big performance boost? The test results clearly indicate that in this field of application this is not to be expected. The performance using first, second and third order Markov is even slightly higher than in the extended case with first, third and fifth order Markov, as can be seen in *Table 3*. This is most likely due to the fact that the usage of up to ten workspaces does not result in very long workspace selection patterns when the user's work is controlled by a workflow. Hence the higher ordered Markov's are too powerful and thereby risk failing by trying to find longer non-existent patterns instead of using the common smaller patterns.

Table 3. Prediction ratio averages from all of the simulation tests, using different configurations of PPM.

Configuration	Prediction Ratio
1st, 2nd and 3rd order Markov	59.9%
1st, 3rd and 5th order Markov	58.1%

7.4.1.3 TDAG

The simulation tests result in manageable profile sizes in both of the TDAG configurations. Both configurations are indifferent in the time perspective. However as in the case as above there is no need to take more history into consideration when making a prediction in these situations. This is confirmed by the results in *Table 4*, both profiles result in approximately the same performance. It is possible that the extended TDAG could perform slightly better if the training session was longer, but this performance boost is not likely to be worth the increased profile size.

Table 4. Prediction ratio averages from all of the simulation tests, using different configurations of TDAG.

Tree Depth	Prediction Ratio
4	63.3%
6	62.8%

7.4.2 Algorithm Properties

Results and analysis of specific properties tested by the artificial tests are presented in the following subsections.

7.4.2.1 Adaptability

When the user changes between tasks and/or alters behaviour this introduces changes in the workspace selection patterns. The algorithms handle this with varying success. *Figure 20* shows that Markov needs more time to adapt to a new pattern, while IPAM very quickly identifies the new pattern and starts to make correct predictions. TDAG and PPM find the new pattern somewhat slower but still rather fast.

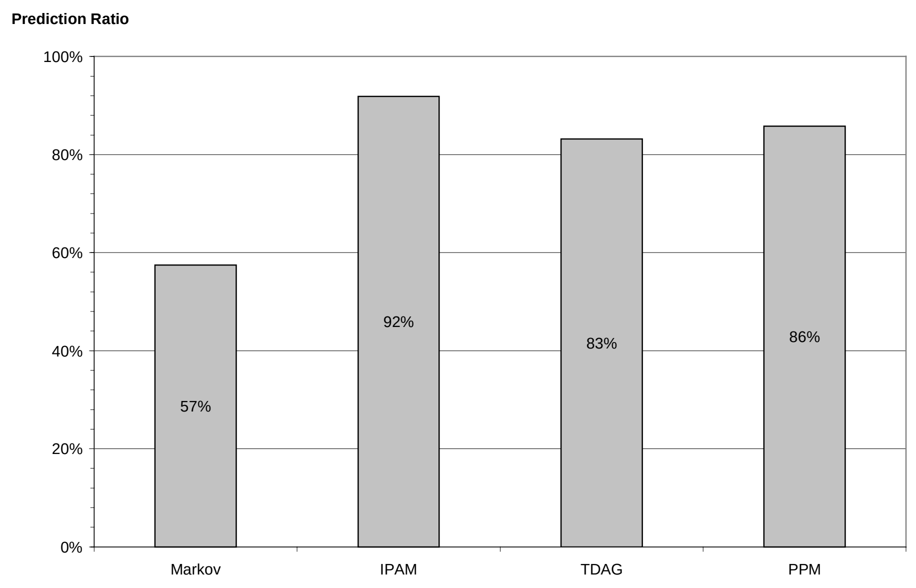


Figure 20. Average results of the adaptability tests.

Markov is not very dynamic. When it has been initialised with a pattern a number of times it requires at least as many feeds with the new pattern to adapt to this. IPAM avoids this by giving higher priority to the most recent data, making it very dynamic.

7.4.2.2 Noise Sensitivity

When the user combines regular workspace selection patterns with total randomness, all of the algorithms naturally are affected in terms of performance.

In the easier tests using only one pattern none of the algorithms, except IPAM, are noticeably more sensitive to noise than another. The prediction ratio naturally goes down during the noise, but they recover immediately as the noise is replaced by a known pattern. IPAM is however not as good as the others. It seems as the adaptability of IPAM makes it more sensitive to noise. IPAM replaces its earlier gathered knowledge with bad data collected from the noise and when the pattern re-appear it makes wrong predictions at first, see *Table 5*.

Table 5. Average results of the simple noise sensitivity tests.

Algorithm	Prediction Ratio
TDAG	100,0%
IPAM	86,4%
Markov	100,0%
PPM	100,0%

In the advanced tests using several different patterns, the performance of the different algorithms varied noticeably. As can be seen in *Figure 21* the situation becomes difficult for both Markov and IPAM. Algorithms based on first order Markov cannot distinguish the patterns from the noise and thus if the noise sequence is long enough they tend to unlearn. Markov can endure noise longer than IPAM since it is not as dynamic but when its limits are reached the same property makes the recovery slower. PPM makes the fastest recovery and starts to make correct predictions earlier than the others. The reason for this is that during the noise the low quality of the predictions saves the knowledge stored in the higher ordered Markov's, since the lower ordered data structures are used more frequent. TDAG also works reasonably well.

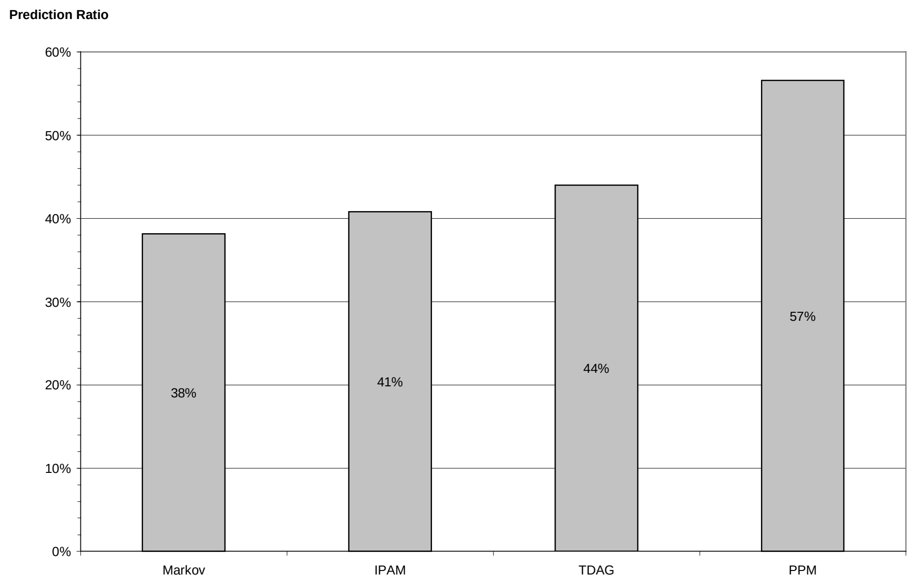


Figure 21. Average results of the advanced noise sensitivity tests.

7.4.2.3 Handling of Irrationality

If the input to the algorithms is pure noise, none of them are able to make any good predictions. As *Figure 22* shows, this kind of irrational usage of the system gives only slightly shorter waiting times than a system without prediction mechanism. The prediction ratios are close to the performance that should be reached with random pushing, 1/7 ≈ 14.3%, varying from 11.6% to 14.2%. An important observation is that none of the solutions result in worse waiting times than in the case where predictions are not used, even if the input only contains noise. This could easily be interpreted to mean that the prediction mechanism never can decrease the performance of the system, this is however not entirely true since the unnecessary load on the network could result in longer waiting times in large systems.

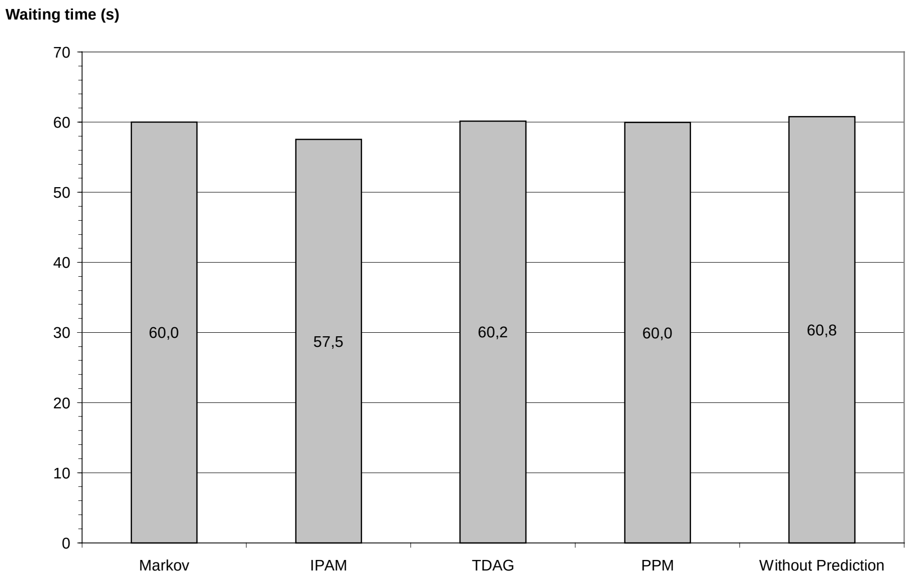


Figure 22. Average waiting times in the irrationality tests.

7.4.2.4 Ability to Identify Long Patterns

When the algorithms are used with several similar long patterns there are two contenders that distinguish themselves, see *Figure 23*. These are TDAG and PPM. Both of these are configured to recognise patterns of length four, even if most of the patterns used in these tests exceed this length these algorithms perform well. Since Markov and IPAM base their predictions on only the current and the previous workspace their ability to identify long patterns are very limited, as can be seen in the test results. The performance of this type of algorithm decreases drastically as the number of different patterns and the lengths of the patterns increase.

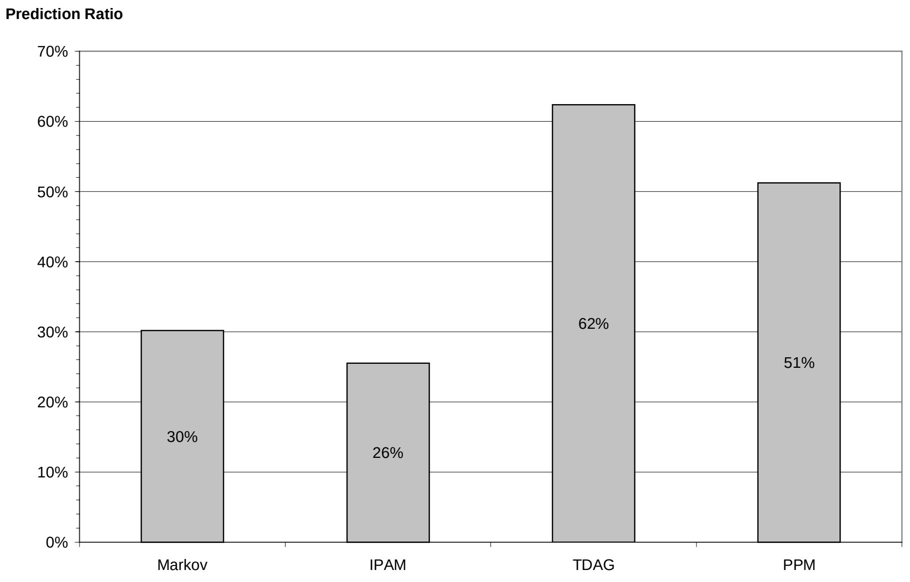


Figure 23. Average results of the long pattern tests.

7.4.3 Algorithm Performance

The algorithm performance is evaluated by measuring the prediction ratios in the different simulations.

The results from the library system simulation are shown in *Figure 24.* In this test TDAG gives the highest performance. PPM also performs well. The others fail to produce acceptable prediction ratios. The reason for this is that this test uses several, similar patterns of approximately the same length. This requires the algorithms to be able to identify and separate long patterns, a property that Markov and IPAM lacks.

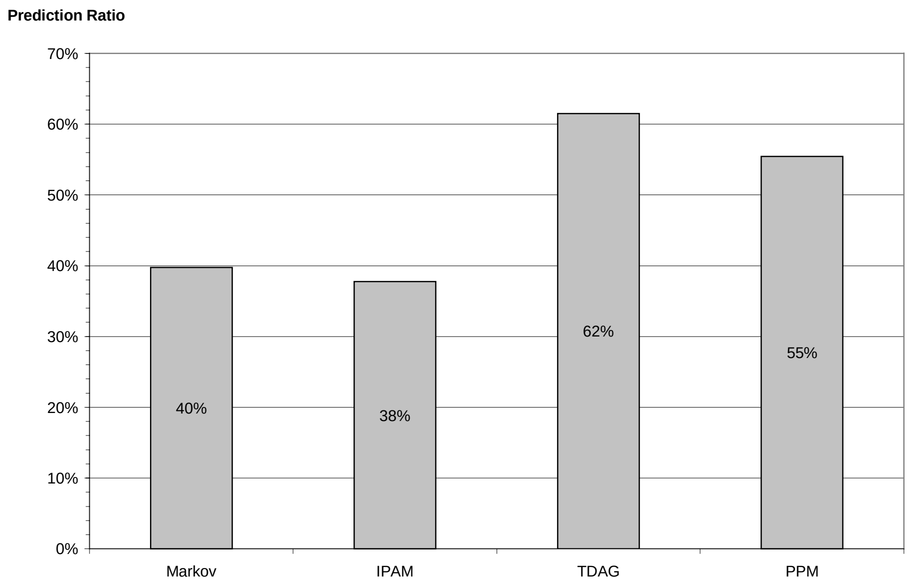


Figure 24. Prediction ratios achieved in the library system simulation.

The SKL simulation contains three subtests, the result from each of these are shown in *Figure 25*. In both the administrator and the regular user tests TDAG turns out to have the best performance. However all algorithms perform almost as good, the reason for this is that these tests includes a lot of short patterns due to the way PAX-E systems are designed. It is a good sign that the administrator test with its sudden role changes did not result in unacceptable prediction ratios.

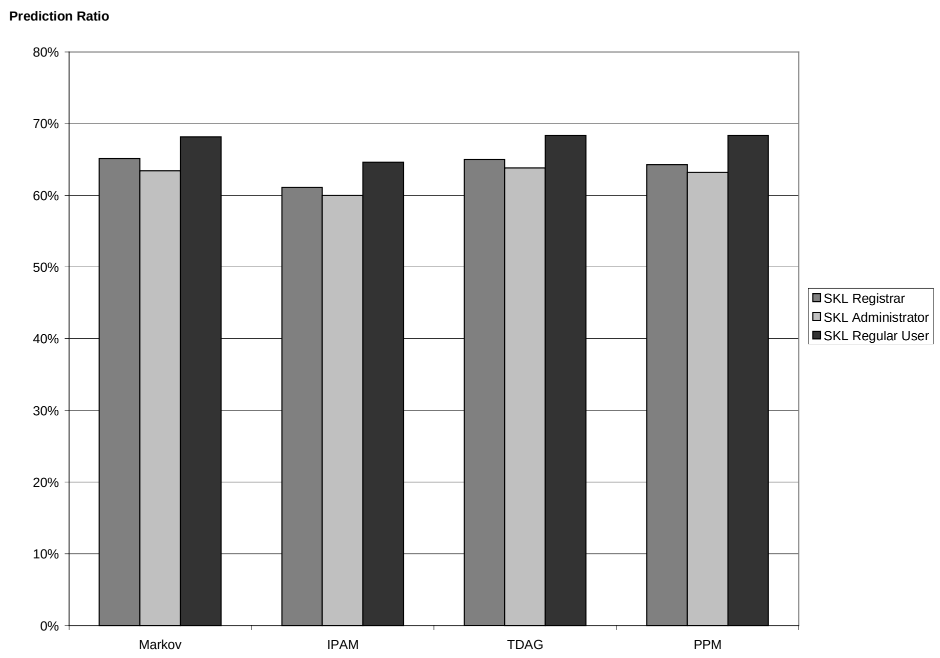


Figure 25. Prediction ratios achieved in the SKL simulations.

TDAG has a good prediction ratio in all of the simulations, closely followed by PPM. Although Markov and IPAM perform unexpectedly well their ratios drops drastically in the library system simulation and they cannot keep up with the other two contenders.

By analysing the server-side logs from the simulations the required training duration for the different algorithms can be estimated. This show how long the push should be disabled during the initialisation of a subprofile, see *5.4.2.1 Initial Push Configuration*. For a subprofile to be trained enough it should at least provide a prediction ratio of 50%. Otherwise it is hard to justify the usage of pushing, an on-demand solution with a large cache is probably better in situations where a subprofile cannot make the correct choices in at least half of the invocations.

In the library system simulation TDAG required a training phase of length 200 to reach 50% accuracy. PPM was a bit slower, requiring a training phase of length 250. Markov and IPAM never reached the prediction ratio threshold. In the SKL simulations IPAM only needed a training phase of length 30, followed by Markov that required 50. TDAG was a bit slower needing 70 and PPM needed 80.

In all of the simulations all of the algorithms perform updates and prediction making very fast, in fact so fast that this time is insignificant. In terms of required memory all of the algorithms result in more than manageable subprofiles, sizes ranging from 625 bytes to 45 KB.

7.4.4 Cache Performance

The cache performance is evaluated by measuring the hit ratios in the simulations. The results are shown in *Figure 26*.

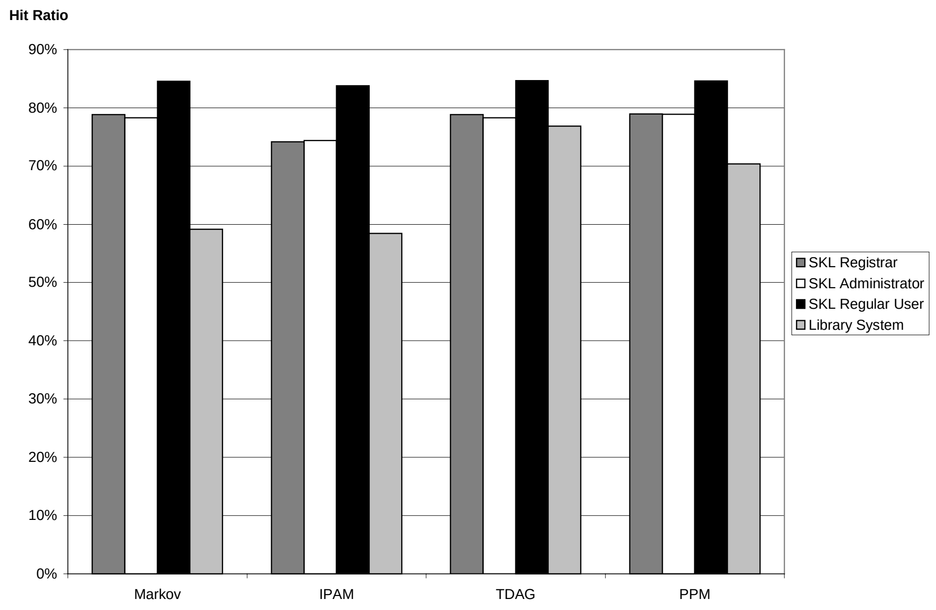


Figure 26. Cache performance in the simulations.

An average hit ratio of 77% is more than adequate and shows that the cache works well together with automatic component transfers.

When the simulations were run without pushing, the cache hit ratio decreased by nearly 40%. This shows the necessity of the prediction mechanism at the server.

7.4.5 Overall Performance

The waiting times when running the simulations with the different solutions are presented in *Figure 27*. Reference waiting times, measured by running simulations without a cache and with disabled pushing, are also presented for comparison.

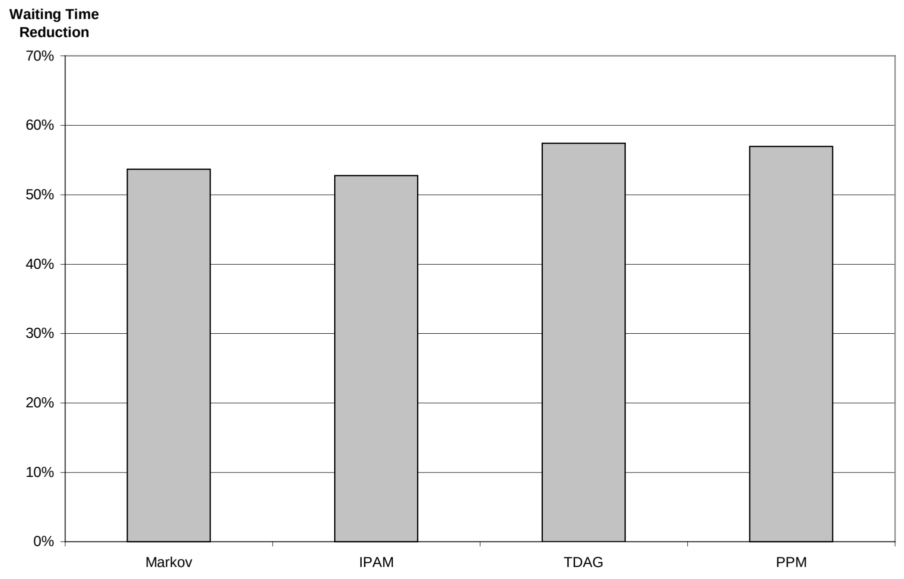


Figure 27. Average performance achieved in the simulations.

As can be seen the most successful solution, based on the distribution model, uses TDAG. By using this solution, the waiting time can be decreased by 58%.

8 Conclusions and Further Work

This chapter presents the conclusions of the examination project by discussing the performance of the tested algorithms and the distribution model. Also some ideas for further work are presented.

8.1 Algorithm Choice

Several algorithm properties were evaluated in the implemented testbench. The results are summarised below.

- Adaptability. IPAM is best at adapting itself to new situations. TDAG and PPM are somewhat slower. The adaptability of Markov is inferior to the other algorithms.
- Noise sensitivity. PPM and TDAG are less sensitive to noise than Markov and IPAM.
- Handling of irrationality. The algorithms are all affected bad at handling total randomness.
- Ability to identify long patterns. TDAG and PPM are better at identifying long patterns than Markov and IPAM.

TDAG and PPM have the best properties of the proposed algorithms. Although IPAM is more adaptable than these algorithms it suffers from higher noise sensitivity, which is not wanted in the distribution model.

The performance of the different algorithms in real systems was evaluated in two types of simulations. In the simulation based on usage of the implementation TDAG had the highest prediction ratio. PPM also performed well while Markov and IPAM failed to produce acceptable ratios. In the simulation based on a PAX-E system delivered to SKL this was not the case. Once again TDAG had the highest prediction ratio, but this time Markov came second. It was closely followed by PPM. The prediction ratio produced by IPAM was noticeably lower, though still acceptable. The required training duration for the different algorithms varied a little. TDAG is a bit faster than PPM at acquiring enough knowledge to be permitted to start pushing. The other two algorithms failed the library system simulation, but in the SKL simulations they were twice as fast as TDAG and PPM.

All algorithms are fast at making predictions and updates. None of them require a large amount of memory or an exceptional amount of training. However they are not as similar in terms of performance and abilities. Two contenders proved themselves to be more suitable than the others. These are TDAG and PPM. Since TDAG performed best in the simulations it seems to be the best choice. It is also highly configurable, making it possible to fine-tune it. In addition to this it does not have to rebuild its data structures when a user’s privileges are altered, as in the case with the other algorithms.

8.2 Usage of the Distribution Model

The performance increase reached by incorporating the model in a system with thin clients depends on the system characteristics. However it has been shown that the waiting times can be expected to decrease by nearly 60% in a system suitable for user prediction.

The advantages of the distribution model are, besides the decreased waiting times, that it is very easy to implement and that it does not increase the amount of needed administration work. The price for this is increased network and server load. However if the model is combined with well-implemented traffic shaping the network performance does not necessarily have to decrease perceptibly. The possibilities with traffic shaping were discussed in *3.3.1 Introduction to Push*. If the server load reaches unacceptable levels, the work can be shared between several servers by using load balancing.

The SKL simulation was heavily based on a real world PAX-E solution. The results of those tests clearly prove that PAX-E is not only suitable to use together with user prediction, but also that satisfactory waiting times can be achieved without requiring a large amount of memory at the client. The simulations also show that the length of the training phase, where the pushing is disabled while TDAG builds a sufficient tree to base predictions on, must be decided from system to system. A system where the users tend to work in many selection patterns of equal length, as in the case with the library system simulation, seems to require long training phases.

Even if the distribution model was designed particularly for PAX-E it should also work well in other type of systems, but such a system must fulfil the requirements mentioned in *3.2.1.4 Similarities Between the Areas*. Also the system must be analysed thoroughly in order for the developers to be able to configure TDAG properly.

The conclusion is that the distribution model indeed decreases the waiting times and that the disadvantages can be handled by supplementary solutions.

8.3 Fulfilment of Requirements

Requirements of the distribution model listed in *2.3 Requirements* are fulfilled in the following way:

- The model is designed for distribution of server based components.
- The model is suited for PAX-E, since the basis of the model is the workspace and component model used in PAX-E. Customisations, like component sizes and number of workspaces, have been made in accordance to current PAX-E systems and future PAX-NG systems. The model can be configured to work in other types of system, as long as these are suitable to combine with user prediction based on discrete sequence prediction.
- The model requires no extra administration. The only manual configuration of the profiles consists of numbering the user’s workspace privileges. This configuration must be made in a PAX-E system anyway.
- The model can be implemented platform independently in languages such as Java. It also permits different types of clients to be used.

8.4 Further Work

Some ideas about further work around the distribution model presented in this thesis are presented in the following subsections. The most important studies that remain are an extensive security analysis and an evaluation of the scalability of the distribution model.

8.4.1 Neural Networks

Neural networks are a learning mechanism used in the area of artificial intelligence. A neural net is input with data over and over again and thereby learns from it. After this initialisation, the net can be used to extract knowledge from. This might be used to base user predictions on. The drawbacks with neural networks are that they require a lot of training data before any information of sufficient quality can be extracted. Most neural networks are only initiated with data during the learning phase and therefore no incremental updating is possible. However there are some special variants that allow incremental updating. These might be used in a prediction mechanism and therefore a deeper evaluation of the area of neural networks could prove itself useful.

8.4.2 Security Analysis

In most systems, security risks exist and often cause a lot of trouble. The security of the distribution model should therefore be analysed. The increased amount of network communication and the way this communication is performed might introduce security problems that have not been realised.

8.4.3 Scalability

The scalability of a system incorporating the distribution model is an interesting property that has not been possible to test in this examination project. In the test implementation only one user at a time can be logged in. Performance tests of a system, extended to allow several concurrent clients, are important to make before using the distribution model further. How the distribution model works together with the supplementary solutions mentioned in *8.2 Usage of the Distribution Model* should also be evaluated.

8.4.4 Comparison of CORBA and RMI

The total performance of the implemented system can be increased if the communication between the server and the client can be speeded up. Even though there are several tests of CORBA and RMI in reports and books, an extensive evaluation of which of these that is best to use in a professional server based component system should be made. An interesting question is how much can changing the communication from RMI to CORBA speed up the system? What is the cost in develop time for this increased performance?

8.4.5 Fine-tuning of Algorithm Effectiveness

Another idea about further work is to evaluate the possibility to fine-tune and increase the performance of the prediction algorithm. The height of the tree used by TDAG can be increased, if the target system seems to require this. To save memory in such a situation pruning of seldom visited nodes can be implemented. If the size of the tree is increased, limiting the size of each prediction can shorten the time it takes to make a prediction. A higher threshold value for `out-count` might result in a higher performance in some systems. This change will extend the learning phase of the algorithm, but when it starts to make predictions these should have higher quality.

8.4.6 Activation and Deactivation

The perceived waiting times that have been decreased by the introduction of a push mechanism could be reduced even more if the activation and deactivation of the components are made in advance. This could be implemented by extending the clients to allow recently pushed components to pre-activate themselves in the cache. When a prediction is correct the time to plug in the component into the GUI framework thus should be drastically reduced.

References

[Allen et al. 1998]	Allen, C., Kania, D. and Yaeckel, B. *Guide to One-To-One Web Marketing.* New York: John Wiley & Sons Inc. 1998. ISBN 0-471-25166-6.
[Bladh 1995]	Bladh, M. *User Modelling in a Help System for a System Development Process*. Linköping: Department of Computer and Information Science at Linköping University 1995. ISRN LiTH-IDA-Ex-9507.
[Çentimentel et al. 1999]	Çentimentel, U., Franklin, M. J. and Giles, C. L. *Flexible User Profiles for Large Scale Data Delivery*. Maryland: Institute for Advanced Computer Studies at the University of Maryland 1999.
[Cerami 1998]	Cerami, E. *Delivering Push*. New York: McGraw-Hill 1998. ISBN 0-07-913693-1.
[Coulouris et al. 1994]	Coulouris, G., Dollimore, J. and Kindberg, T. *Distributed Systems Concepts and Design*. Second Edition. Essex: Addison-Wesley Longman 1994. ISBN 0-201-62433-8.
[Crovella & Barford 1997]	Crovella, M. and Barford, P. *The Network Effects of Prefetching*. Boston: Computer Science Department at Boston University 1997.
[Curewitz et al. 1993]	Curewitz, K. M., Krishnan, P. and Vitter, J. S. Practical Prefetching via Data Compression. *The 1993 ACM SIGMOD International Conference on Mangagement of Date*, Washington, USA, pp. 257-266, May 1993.
[d-tec 1998]	Distributed Technologies 1998. *3-Tier Architectures.* Accessed 1999-11-10. http://www.corba.ch/e/3tier.html
[Davison & Hirsh 1997]	Davison, B. D. and Hirsh, H. *Experiments in UNIX Command Prediction*. New Jersey: Department of Computer Science at Rutgers, The State University of New Jersey 1997.
[Davison & Hirsh 1998]	Davison, B. D. and Hirsh, H. *Predicting Sequences of User Actions*. New Jersey: Department of Computer Science at Rutgers, The State University of New Jersey 1998.
[Donkers 1997]	Donkers, H. H. L. M. *Markov Decision Networks*. Maastricht: Faculty of General Sciences at Maastricht University 1997.

[Edlund 1998]	Edlund, P. Distribuerade objekt med Java, CORBA vs DCOM vs Java RMI. Linköping: Department of Computer and Information Science at Linköping University. ISRN LiTH-IDA-Ex-98/9.
[Engberg 1998]	Engberg, T. *Master's Thesis: Distributed HTTP cache*. Luleå: Luleå Tekniska Universitet 1998. ISRN LTU-Ex-98/359-SE.
[Horstmann & Cornell 1998]	Horstmann, C. S. and Cornell, G. *Core Java*$^{1.1}$ *Volume II – Advanced Features*. California: Sun Microsystems Press 1998. ISBN 0-13-766965-8.
[Juric et al. 1998]	Juric, M. B., Rozman, I. and Zivkovic, A. Are Distributed Objects Fast Enough? *Java Report*, vol 3, no 5 pp. 29-38 & 65, 1998.
[Kauffman 1997]	Kauffman, T. 1997. *3-Tier Client/Server Research Page*. Accessed 2000-01-19. http://sandbox.aiss.uiuc.edu/3-tier/index.htm
[Laird & Saul 1994]	Laird, P. and Saul, R. Discrete Sequence Prediction and Its Applications. *Machine Learning*, vol. 15, no. 1 pp. 43-68, 1994.
[Lau 1999]	Lau, T. *A comparison of sequence-learning approaches: implications for intelligent user interfaces*. Washington: Department of Computer Science and Engineering at the University of Washington 1999.
[Luotonen & Altis 1994]	Luotonen, A. and Altis, K. 1994. *World-Wide Web Proxies*. Accessed 2000-01-19. http://www.lib.uwaterloo.ca/IRC/Annual_Report/wropresent/Overview.html
[Marimba 1999]	Marimba 1999. *Marimba Products*. Accessed 1999-11-10. http://www.marimba.com/products/products.htm
[Microsoft 1997]	Microsoft 1997. *DCOM Architecture*. Accessed 2000-01-19. http://msdn.microsoft.com/library/backgrnd/html/msdndcomarch.htm
[OMG 1999]	Object Management Group 1999. *News & Info*. Accessed 1999-11-10. http://www.omg.org/corba/index.html
[Russell & Norvig 1995]	Russell, S. J. and Norvig, P. *Artificial Intelligence A Modern Approach*. New Jersey: Prentice-Hall 1995. ISBN 0-13-360124-2.
[Shaheen 1999]	Shaheen, T. *What is Caching?* Accessed 2000-01-19. http://dimacs.rutgers.edu/~tshaheen/caching.html

[Sun 1999/1]	Sun Microsystems 1999. *Java™ Remote Method Invocation*. Accessed 1999-11-10. http://java.sun.com/products/jdk/rmi/index.html
[Sun 1999/2]	Sun Microsystems 1999. *RMI over IIOP*. Accessed 1999-11-10. http://www.javasoft.com/products/rmi-iiop/index.html
[Sun 1999/3]	Sun Microsystems 1999. *JavaBeans^{TM} Architecture: Infobus*. Accessed 2000-01-07. http://java.sun.com/beans/infobus/index.html
[Sun 1999/4]	Sun Microsystems 1999. *Enterprise JavaBeans^{TM} technology*. Accessed 2000-01-19. http://java.sun.com/products/index.html
[Szyperski 1998]	Szyperski, C. *Component Software Beyond Object-Oriented Programming*. Essex: Addison Wesley Longman 1998. ISBN 0-201-17888-5.
[Wong 1998]	Wong, A. *Prophet: Predictive Profiling for Web Cache Eviction*. Cambridge: Computer Science Department at Harvard University 1998.

Contents

Figures

Tables