On balancing client - server load in intelligent web-based applications involving dialog

Michelle Doyle, Dr. Pádraig Cunningham.
Department of Computer Science, Trinity College Dublin 2, Ireland.

Abstract

With the explosive growth of the Internet have come problems of increased server load and network latency. This means that systems that require interchange of data between server and client over the network can be slow to unusable (at peak times). We are investigating methods for developing distributed CBR systems which *minimise* the load on the servers and the network, thus increasing response time and usability. The first part of this research focuses on balancing load in a client/server situation that involves a long-lived dialog. This report discusses the various issues to be tackled when attempting to effectively balance the processing load.

Introduction

This report summarises the work to date in one strand of the CBR-Net project. This project is in the area of networked/distributed Case-Based Reasoning. Case-Based Reasoning (CBR) is a methodology found in Artificial Intelligence, which takes solutions to problems encountered in the past as a basis for solving current problems. The research focuses on networked CBR systems, particularly those to be found on the Internet. This report assumes the reader is familiar with the basics of CBR, and will give only a brief introduction to the area, concentrating instead on the particular problems involved in networked CBR systems.

With the explosive growth of the Internet have come problems of increased server load and network latency. This means that systems that require interchange of data between server and client over the network can be slow to unusable (at peak times). We are investigating methods for developing distributed CBR systems which *minimise* the load on the servers and the network, thus increasing response time and usability.

This paper presents the current state of the research and possible future directions. The original objective of the research was to design and develop CBR tools that would support the development of distributed CBR applications on Inter- and Intranets. So far a generic client-server shell has been developed in Java which supports incremental CBR applications and tries to balance the load between the client and the server according to the current network conditions. Since this shell is

domain-independent, it could theoretically be used with any Case Base (simply by supplying the data files in the correct format), but the nature of the incremental engine currently limits us to Case Bases with discrete feature values and solutions. This is a solvable problem, however.

The above topics are considered in more detail in the following sections, beginning with a brief introduction to CBR and particularly incremental CBR (section 1). Reasons for wishing to move away from the usual "thin" client - "fat" server approach are discussed in more detail in section 2. Section 3 looks at the development of the current system, mentioning the issues that emerged during this development. These issues are discussed in greater detail in section 4. Section 5 presents conclusions and some possible future directions.

1 Case-Based Reasoning

1.1 Introduction to Case-Based Reasoning

"A case-based reasoner solves new problems by adapting solutions that were used to solve old problems" [Riesbeck & Schank 89]. Case-Based Reasoning is an AI approach to solving problems that attempts to solve the current problem by relating it to past experiences (cases). The intuition behind CBR is that situations often reoccur and using the knowledge gained from solving a similar problem in the past is a good starting point from which to solve the current problem. As there are usually some differences between the target (input) problem and the retrieved case(s), some sort of modification may need to be applied to the ballpark solution suggested by the old case, so that it fits the new situation. This is known as adaptation. Cases that were solved successfully can be retained and so are available for further use. Failures can also be retained, giving the reasoner the ability to recognise potential problems so that they can be avoided in the future. Both of these processes result in the reasoner's knowledge constantly evolving - thus it learns as a natural by-product of problem solving.

Part of the reason for the interest in CBR is its psychological plausibility. It can be seen at work in many everyday situations - previous experiences are often utilised in decision-making processes. When trying to plan a big night out (to celebrate your birthday, for example), with a gang of friends with varied personalities, you need to remember previous good and bad experiences on nights out with these people, in order to successfully plan a night that everyone enjoys.

CBR has many advantages over traditional rule-based systems. Remembering previous situations similar to the current one in order to solve the new problem is a more plausible approach to problem solving than always reasoning from first principles. This is clearly a more efficient approach as well. As well as this, CBR can also be used in domains where no causal model is available, as the reasoner does not need explicit rules or models to reason from - it makes use of the implicit knowledge contained in cases. This also means that CBR does not suffer from the knowledge elicitation problems that characterise expert system development (known as the *knowledge elicitation bottleneck*).

The generally accepted CBR cycle is: Retrieve, Reuse, Revise, and Retain [Aamodt & Plaza 94]. The system described in this paper has a slightly different sequence of events – it uses a process of *incremental* retrieval, where the initial query does not have to be completely specified; instead it is used to retrieve an initial set of cases that is incrementally narrowed down through further queries.

1.2 Incremental CBR

The approach to incremental CBR (I-CBR) used in this system was proposed by Smyth & Cunningham (see [Cunningham & Smyth 94], [Smyth & Cunningham 95]). The idea was originally proposed for particular types of diagnosis problems, where it is difficult to gather a complete case description in advance. In these types of problems, there is a potentially large amount of information that could be used to aid the diagnosis, but all of this information is not necessarily needed to solve the problem. Moreover, some of this information is “expensive” - meaning it may be difficult or costly to acquire - and so it is desirable to minimise the use of these features. A multi-stage retrieval process achieves this. The “free” (or readily available) features can be input to the initial (under-specified) query. This “first pass” retrieves a subset of the initial Case Base, and subsequent refining questions reduce the size of the retrieved set even further, until a consistent set of cases remains.

The most important component of the I-CBR approach is how the “refining questions” are chosen. An information theoretic approach is used, which chooses the next feature based on its discriminatory power (or "information content") with regard to the current set of retrieved cases (see [Smyth & Cunningham 95] for a detailed description). In simple terms, this approach measures *how much information is gained* about the possible class of the target case *through knowing the value of the feature*. This method of choosing refining questions makes the reasoning more focused, and has been shown to out-perform a goal-directed reasoner in terms of number of questions posed ([Smyth & Cunningham 94]).

This information theoretic measure is essentially the same as that used in ID3 [Quinlan 86] when selecting the best attribute at each step in growing the decision tree. However, with I-CBR, this discriminatory power is assessed in relation to the current set of retrieved cases at runtime and not in relation to the entire Case Base. Another important difference is that ID3 is an algorithm for building a full decision tree *offline*, whereas with I-CBR only partial decision trees are built *on the fly* over the current set of retrieved cases.

Unfortunately, I-CBR is also restricted by this approach to choosing the next refining question. In order to assess the discriminatory power of a feature, it is necessary to know in advance

- all the possible values for that feature
- all the possible classifications of cases

This means that I-CBR in its basic form, like ID3, can only handle classification problems and discrete-valued features. However there are various extensions to the basic ID3 algorithm, which attempt to overcome these problems (see [Quinlan 93]).

As was mentioned previously, I-CBR was originally proposed for diagnosis tasks where it is both costly and unnecessary to completely specify an initial query. However, incremental retrieval could be useful for other tasks also. Take for example a CBR travel advisor, or production configuration system. The user may not have a very specific idea of his requirements when starting to use such a system. With an incremental system, they only need to specify the features most important to them to begin with, and will hopefully only need to answer a subset of the remaining questions in order to retrieve useful cases.

However, decision trees can find different categorisations for inputs depending on the order in which features are input. This means the case that is found is the best possible case for a query where the questions are asked in order of importance. Since the information theoretic approach asks questions in an order that reflects the discriminatory power of that question at that time, questions are not asked in an order that takes the user's view of importance into account.

Despite the restrictions of the basic version of I-CBR, the ability to engage the user in dialog instead of performing *one-shot* retrieval makes it an attractive method of retrieval. However, this dialog may be long lived, and if the retrieval is taking place across a network, server and network load could result in slow response times for the users. For this reason, there is a need to investigate ways to improve response times with such systems.

2 Network concerns

The Internet (or "global information system") is growing at an extremely high rate. Traffic on the web is in fact increasing at exponential rates [Markatos & Chronaki 98]. Even though there have been dramatic increases in the performance of networks and computers, the rapid growth of the Internet, and increasing levels of traffic, make it difficult for Internet users to enjoy consistent and predictable end-to-end levels of service quality [Ferguson & Huston 98]. Increased numbers of users has lead to increased network and server load. At peak times, some services can be completely unusable, as the response from the server is painfully slow.

Ferguson and Huston list the quality of service on the Internet as dependent on four factors – delay, jitter, bandwidth and reliability.

- *Delay*s we are all familiar with. The higher the delays, the more stress is placed on the transport protocol to operate efficiently, as greater amounts of data are being held "in transit" in the network.
- *Jitter* (the variation in end-to-end delay) is also an oft-observed phenomenon. High levels of jitter make round trip time (RTT) difficult to estimate. The TCP protocol expects destinations to send back acknowledgements whenever they receive new data segments, and uses this information to calculate the RTT. If this acknowledgement has not been received before a particular timeout period, the data is assumed lost and retransmitted. The timeout period is calculated based on average RTTs observed. If jitter causes an overly conservative RTT estimate, the timeout will also be too long; meaning it will take an unnecessarily long time before packet losses are detected. On the other hand, overly short RTT estimates will mean a low timeout, causing packets to be retransmitted unnecessarily, which is a waste of network bandwidth.
- The *bandwidth* is the maximal data transfer rate between two end points. This rate does not depend solely on capabilities of the physical network path, but also on the number of other routes sharing common components of the path. Therefore as traffic increases on any of those routes, they all suffer.
- The overall *reliability* of the network can be though of as the average error rate or as an indication of the performance of the switching system. Routing problems can cause increased delay and reduced reliability (through packet loss). Instability in the routing protocols can lead to increased jitter (see [Labovitz et al. 97]).

There continues to be much research into improving the Internet at the hardware and protocol levels in order to increase reliability and tackle the problems of delay and jitter. However guarantees are not possible in the Internet in the foreseeable future

[Ferguson & Huston 98]. We need to look at ways to maximise the reliability and responsiveness of high-level Internet services by *minimising network traffic*. System architectures that *minimise centralised server load* would also increase responsiveness. Such architectures would have the twofold advantage of decreasing processing load on the server (resulting in an increase in the number of available CPU cycles) and also decreasing the network congestion in the vicinity of the server. This kind of architecture can be seen at work in much of the recent research into web caching and prefetching of documents.

Web caching is concerned with caching popular documents close to clients. The aim of this work is to reduce network traffic and server load by keeping data close to clients that re-use them. Prefetching of documents takes this a step further; this approach makes use of client browsing patterns and server statistics to automatically download to clients or proxy caches documents that could be useful in future (see [Markatos & Chronaki 98], and [Hine et al. 98] for two such systems). This "prefetching" could be performed at off-peak times, so that transfer costs would be minimised. This reduces server load and also reduces latency for clients, as the documents are close by when they want them.

Gwertzman [Gwertzman 95] takes another approach to this problem. He proposed the idea of "geographical push-caching" to reduce a web server's load. When the load on a web server exceeds some limit, it replicates ("pushes") the most popular of its documents to other co-operating servers, provided they have a lower load. Clients can then make future requests for these documents from the other servers (hopefully from the server closest to them). This approach reduces server load and bottlenecks close to popular servers. However it does have its problems. Clients may still have to contact the original server to find out where they should get the documents, increasing response time at least slightly. The servers containing the replicated documents might still be far enough away for latency concerns, unlike prefetched documents which reside on the clients themselves or local proxies.

All of this research shows that the current trend is to decentralise load in order to overcome responsiveness and reliability problems on the Internet. It seems a natural progression to consider applying the same approach to client-server systems, especially those involving a long-lived dialog. The traditional client-server architecture consists of a "thin" client (which is little more than GUI) and a server that handles all of the processing. Willcox [Willcox 97] advocates this kind of system, and argues that downloading the amount of code that would be needed by a "heavy" client would be prohibitive. However, unless you have a very expensive high-powered server, server overload is a serious concern with these systems, and either way, network overload can potentially slow down responses from your server. We

also argue that current archive formats can lead to much reduced code size (see section 4.2), making once-off transmission of code feasible, once the time required is offset by the time saved on dialog later.

Therefore our approach has been to move processing from the server to the client as soon as possible. However, Case Bases can be very large, so it would not be possible to send over the Case Base at the beginning. Incremental retrieval is used to prune the Case Base step by step, until the Case Base is small enough to send to the client. The details of how this works in practice are given in the next section.

3 The current system

The implementation stage of this project began by modifying an existing system, which was implemented in Java. This system was a traditional "thin" client system, and is well documented in [Doyle 97]. There were various extensions necessary to make this system incremental and give it the ability to move processing from the server to the client. These extensions are described step by step below.

Stage 1 – Incremental Retrieval

The first step was to change the retrieval mechanism from one-shot to incremental. The underlying retrieval algorithm (spreading activation) was unchanged however. In the original system, retrieval was performed as described below (see [Doyle 97] for more detail):

A `FeatureLinkWeb` object is created when the Case Base is loaded into memory (as a `CaseBase` object). This object is made up of `FeatureLink` objects for *each value* of *each feature* of the cases in the Case Base, and it contains pointers to all the cases with that value for the feature[1]. These objects are used to speed up retrieval. When a target case (T) is input to the system, the Case Base is first partitioned according to its constraints (if any). Constraints are features that *must be matched* in the cases retrieved. Therefore only cases that have the same value as T for each constraint are put into the partition. This stage is known as *Base Filtering*. The *Spreading Activation* stage then proceeds as follows:

[1] Creating all of these `FeatureLink` objects might take a while with a large Case Base. This is taken care of in the current system by only creating the `Casebase` object once, when the servlet is initialised, and keeping it in memory throughout accesses to the server (see section3, stage 3). In the current system the Case Base is never updated, but if it were, extensions could be made to the `FeatureLinkWeb` object quite easily, so there is no need to recreate it from scratch.

For each descriptor d of T:

All `FeatureLink` objects representing descriptors *similar to* d are found[2]

The activation of every `Case` object, which is pointed to by one of these `FeatureLink` objects **and** is inside the partition, is updated by an amount corresponding to the weight of the current feature and their similarity score

Finally, the cases with activation above a pre-determined threshold are returned[3]

This retrieval strategy was also used in the updated system, in conjunction with the I-CBR algorithm, for improved accuracy. The reason for this is simple. If we used the partial decision tree constructed at each step to retrieve cases, then only cases with *exactly* the same value as the current input value would be retrieved. Using the above spreading activation algorithm, cases with *similar* values to the input value are activated also, allowing cases which may only partially match along all dimensions, but are a potentially good solution, to be retrieved also. The updated algorithm is as follows:

Input: Initial Query Q

For each *specified* descriptor d of Q:

All `FeatureLink` objects representing descriptors *similar to* d are found **(1)**

The activation of every `Case` object, which is pointed to by one of these `FeatureLink` objects **and** is inside the partition, is updated by an amount corresponding to the weight of the current feature and their similarity score **(2)**

retrieved_cases ← current set of cases with activation above threshold

While retrieved_cases is inconsistent and more unknown features:

Calculate the most discriminating feature (f) and ask user

d ← f + value input by user

Perform steps **(1)** and **(2)** above with d as input

retrieved_cases ← current set of cases with activation above the threshold

Finally return retrieved_cases if it is consistent or if all features have been asked

[2] The Case Base administrator determines the similarity threshold. If similarity is less than one, more cases are activated (by an amount relative to their similarity), making the system more accurate as it handles partial matching.

[3] The cases that are returned may additionally have to go through an adaptation phase, depending on the retrieval mode.

The preceding algorithm leaves out the details of one important step – calculating the most discriminating feature. This step uses the information theoretic measure mentioned in section 1.2. It is a direct implementation of the algorithm in [Smyth & Cunningham 95] and I will not go into the details of it here.

Implementing this retrieval algorithm also required modifications to the client side of the system. The client side was originally just an applet, which reacted to certain actions and sent and received data over a socket connection. In order to facilitate asking further questions as the feature names were sent over from the server, a separate thread was created to handle further user input. This was necessary, as waiting for the user to enter a value for the feature being asked is the same as suspending the current thread. This is fine if the current thread is a thread created specifically for this task and has nothing else to do. However, if this loop awaiting input were inside the applet, this would result in a suspension of the thread which invokes the "action" and "handleEvent" methods on components, and basically block all user input (including input in the component which takes in the current value being asked).

This separate thread was also used to handle communications over the socket, as this resulted in a neat separation between the interface and the "front end" of the client system. This separation would be needed for stage 2.

During this stage the existing code (which had been developed using JDK 1.0.2) was also updated to be JDK 1.1 compliant.

The next step was to implement an initial “fat” client version of the system.

Stage 2 – The initial “fat” client system

As mentioned above, the existing system was a traditional client-server system, with only a GUI and code for action events and socket communications on the client. The aim was to extend this system to conform to the architecture depicted in Figure 1.

Since the interface had already been separated from the dialog thread, there was no need to make changes to the applet. The dialog thread had to be updated to understand when the Case Base had been transferred, and to communicate with the client-side CBR engine from then on. Therefore the applet is completely unaware when the processing moves to the client side as it only performs display functions.

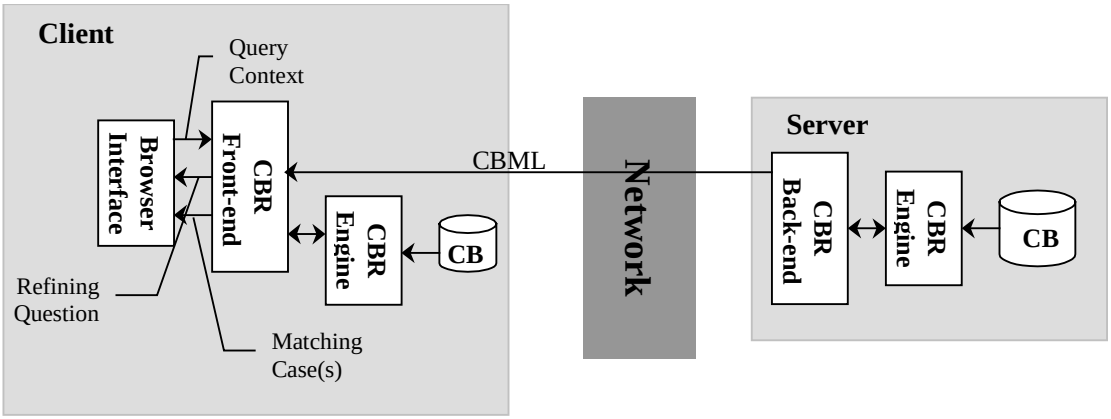

Figure 1. Proposed architecture for distributed CBR

The next issue to be faced was how to download the application logic to the client so that processing could take place there after the Case Base is transferred. Transferring the classes in the background while the applet is processing the initial stages of the query would be ideal, but to date no efficient way of achieving this in Java has been found. The only way of transferring the code in the background is through use of the `Class.forName(`*`classname`*`)` method, which initiates the loading of a class[4]. Therefore you can write a separate thread which takes a list of all the classes in your system, and loads them one by one. This would not be a great solution however. Apart from the need to update this list every time the implementation changes, this approach would be very inefficient as the uncompressed bytecode could be quite large. What is needed is some way to load a JAR (**J**ava **AR**chive) file in the background, but this is currently not possible. Therefore we decided to simply download a JAR file that contains all of the classes needed (including the applet and dialog thread) at the beginning, as at least then the bytecode is compressed and may not be prohibitively large. This issue is discussed further in section 4.2.

In the old system, there is a server program listening on a particular port for client connection requests, and for each request, a socket connection to the client is opened and a `cbrServerThread` object created to deal with this client. Therefore changes needed to be made to the `cbrServerThread` class so that it not only marshalled information between the client and the retrieval engine, but could also decide to send the Case Base across to the client at an appropriate time.

The difficult issue to be faced at this point was how to decide the time was right for transfer of processing. This is not an easy problem (with some sub-issues), and has still not been solved completely satisfactorily. There are a few parameters to be considered:

[4] This is because classes are first loaded in Java when they are referenced.

- The current network latency
- The size of the Case Base
- The number of features yet to be asked

The only one of these that is immediately available is the size of the case base. Estimating the network latency is a very difficult task in itself, even for network layer software. This is discussed in more detail in section 4. For this initial system, a simple calculation of latency was made. A case was sent to the client and back a number of times and the average round trip delay calculated. This value was then divided by two, to give an estimate of the time it would take to send one case from the client to the server. This estimate assumes the delay was the same on each leg of the trip, which may lead to false estimates. However there's no other way to estimate one-way delay without both computers having synchronised clocks. Improvements to this estimation are discussed in section 4. The time it takes to send a feature across was also estimated (for comparison), in a similar way. Every time a refining feature was sent to the client, it was returned to the server immediately as a kind of "acknowledgement". This was used to calculate the current feature transfer time.

The number of features yet to be asked is also not easily calculated, however it may be important. It is difficult to know for sure if sending processing over to the client side is justified, without a feel for how much time would have been spent on dialog over the network. If you have a situation where only 2 or 3 features are asked, it is might not be worth while sending over the Case Base to the client side, as this transfer could take much longer than the few queries. Therefore we need some way of estimating the typical amount of input features needed to find a suitable case within the current Case Base.

It seems possible that there could be a correlation between (Case Base size + number of features per case + number of possible outcomes) and the average number of input features needed to find a solution. This would require quite a lot of experimentation to prove or disprove, however. We might expect that the larger the Case Base, and the more features and outcomes, the better a Case Base would be suited to this type of retrieval, but it would be nice to have some sort of suitability metric. However such a metric might not be easy to devise. Therefore our only option was to find an average for the current Case Base by running the CBR system. Informal tests with the current Case Base[5] produced an average of 10; therefore this value was used at this stage of the implementation. In future, the method used will be to start with a sensible guess,

[5] We are using the 'Large Soybean Database', taken from UCI Machine Learning Repository (http://www.ics.uci.edu/~mlearn/MLRepository.html). This Case Base was edited to remove any cases containing unknown feature values (for simplicity) and now contains 266 cases, each with 35 features, and there are 15 different classes.

record the number of features asked each time the system is run, and automatically update the average each time. That way the estimate improves over time.

With the above estimates for case transfer time and average number of input features, the threshold condition for sending over the Case Base was set to:

```
(avg_case_transfer_time * num_retrieved_cases) <
((avg_input_features - num_features_asked)[6] * feature_transfer_time)
```

It should be noted that no matter what the outcome of the above calculation, the Case Base is not transferred if there is only one feature currently unspecified or if the Case Base is found to be consistent (or *inconsistent*, but all features have already been asked). In these cases, processing is completed on the server side, and results sent to the client.

At this stage some informal tests were run to get a feel for how long the Case Base was taking to download and the improvement in responsiveness once it was. However it was then decided that it might be better to make the final large-scale change (rewriting the system to work with servlets) before conducting proper tests.

Stage 3 - The servlet version of the system

Servlets are modules of Java code that run in a server application to answer client requests. Servlets can only run on servlet-enabled web servers[7]. The current system is running on a Jigsaw web server[8], which in turn is running on the departmental web server so that it can be accessed externally (the college network is behind a firewall). In future this could perhaps be migrated to a Windows NT server which is outside the firewall.

The initial reason for wishing to write a servlet version of the system was because servlets work through firewalls. This gives them the advantage over RMI and sockets. RMI in particular requires a lot of set-up on the server side and is not as efficient as the others. Sockets provide a low-level efficient solution to inter-process communication; however they cannot work through firewalls, as most firewalls do not

[6] Since this subtraction returns 0 if the number of features asked exceeds the expectations, there is a second clause which defaults this value to 2 if it evaluates to 0.
[7] An up to date list of servlet-enabled web servers can be found at http://jserv.javasoft.com/products/java-server/servlets/environments.html .
[8] Developed by the World Wide Web Consortium (W3C), it is an object-oriented server written entirely in Java. See http://www.w3.org/Jigsaw/

allow direct Internet Protocol (IP) traffic between the Internet and the internal network they are protecting.

Servlets are not tied to a specific client-server protocol, but they are most commonly used with HTTP. Most organisations behind firewalls have a WWW proxy server running that allows people inside the organisation to access the Web. Since an applet uses the HTTP protocol to connect to an HTTP servlet, access requests can go through the proxy server. A client request is sent as an HTTP POST request, and the response is sent as an HTTP response. In practice, this means a `URLConnection` object is used to connect to the servlet's URL, an `InputStream` is opened to send the POST request, and an `OutputStream` opened to read the servlet's response.

At first it seemed that servlets wouldn't be suited to our needs, as HTTP is a request-response-oriented protocol, and retains no state information between one request and the next. Therefore there seemed no easy way to manage a proper dialog and remember the set of objects pertaining to a particular user from one request to the next. However, another nice feature of HTTP servlets is that they allow you to manage state information on top of the stateless HTTP. Client-side solutions to this problem include using cookies[9] to store information about users sessions on the client browser. Servlets can make use of these client-side cookies to associate a request with a user.

The way it works is quite simple. When a user first makes a request to a site, they are assigned a new session object and a unique session ID. The session ID is stored on the client-side as a cookie. The session ID is automatically included in every subsequent user request, allowing the user to be matched with his/her session object. Servlets can add information to these session objects, or read information from them. When the session is finished (in our case, when processing or results have been sent to the client side), the session object can be invalidated. The Session Tracker automatically invalidates session objects that have no page requests for a period of time (30 minutes by default). When a session is invalidated, the session object and the data values it contains are removed from the system.

A servlet version of the system was therefore implemented, which uses the above session tracking capabilities to maintain state information between requests. The client-side code also had to be changed, to deal with opening a new connection every time it wanted to send an answer to a refining question. This new version of the system has the following advantages:

[9] See RFC 2109 (Cookies) at http://info.internet.isi.edu/in-notes/rfc/files/rfc2109.txt

- Handles multiple users nicely using session objects and session IDs.
- No persistent connection to the server for each session, which may be good for server load particularly if session is long-lived.
- Because Web-based servlets respond to HTTP methods such as GET and POST, servlet-based communication is able to get around firewalls, which block sockets and RMI.
- Each servlet session spawns a new **thread** (**not** a new process), and it also stays in memory between requests, making it start quickly for each new request.
- Since it stays in memory between requests, it is possible to load the Case Base into memory only once - when the servlet-enabled web server is started. Since the web of `FeatureLink` objects is created each time the Case Base is loaded, it is advantageous not to have to load the Case Base every time a session is started.
- Allows us to access to information about the client, such as their IP address.

At the same time as implementing the servlet version of the system, some of the retrieval objects were redesigned. In the original system, the main class used in retrieval was the `CaseBase` object. This class contained a private `Vector` of `Case` objects, had a constructor which read the cases in from a file and indexed them in a `FeatureLinkWeb` object, methods for finding the most discriminating feature, and retrieval methods such as `firstPassRetrieval(...)`, `refineSet(...)` and `SpreadingActivation(...)`. However, this design meant that multiple users each had an associated `CaseBase` object, which would be a huge memory overhead if there were many users, and already was causing unnecessarily long server connection times (as the `CaseBase` object had to be created at the start of each connection). Therefore, the `CaseBase` class was effectively split into two classes - one (`CaseBase`) which holds the `Vector` of `Case` objects and has the constructor mentioned above, and the other (`retriever`) which maintains an array of activations and an array of indexes to retrieved cases. With this design, every user has an associated `retriever` object, and this object points to cases in the single `CaseBase` object, which resides in memory the entire time the web server is running. This new design, coupled with servlet session tracking, means multiple users can be accommodated efficiently and with ease.

The servlet version does have its restrictions, however. Since there is no persistent connection between the client and the server, the previous method of sending a case back and forth to estimate case transfer time (CTT) is not possible. If a case were sent to the client, the client would have to send a new HTTP POST request to the server to send this case back, and the extra time spent opening a `URLConnection` and setting up data streams would result in incorrect CTT estimates. Therefore other ways of

estimating latency had to be found. This and other problems facing the current version of the system are discussed in greater depth in section 4.

In terms of performance, a direct socket connection is faster than an equivalent HTTP connection, because of the overhead involved in HTTP connections. But using the HTTP protocol is the only reliable way of communicating between an applet and a server through a firewall.

Other improvements

A few other less important changes were made to the system during the various stages. These are mentioned briefly here, before going into the main difficulties involved in trying to implement an efficient load-balancing system in section 4.

- The applet was made truly **generic** by creating the interface on the fly from a supplied "CaseStruct.txt" file.
- During the informal tests of the initial "fat" client system, it was noticed that creating a dialog box every time a user was asked a refining question was not very efficient. When processing was taking place on the client side, there was barely time for the old dialog box to be disposed of before the new one was drawn. This was resulting in slow visual updates for the user, dampening the sense of improved response time and decreasing the general user friendliness. Therefore it was decided to improve the interface before conducting further tests. This was achieved by using different panels and suitable layout managers so only a small sub-panel of the GUI has to be redrawn to ask a refining question. This is more **efficient**.
- Due to a bug in Netscape, the applet `stop()` and `start()` methods are called every time the browser window is redrawn or resized. This should only happen when the user leaves the page, and is a problem as the `stop()` method is supposed to contain code to close connections and stop running threads. Therefore a thread was implemented which monitors whether or not `start()` is called within 5 seconds of `stop()` being called, indicating that this was simply a redraw. All of the code for cleaning up resources (including letting the servlet know the session should be invalidated) was moved to the `destroy()` method, and the “monitor thread” calls this method when it detects the user has actually left the page. If the user *hasn’t* left the page, this thread prevents important processes from being halted or killed. This makes the system more **robust**.

4 The main research issues which have emerged

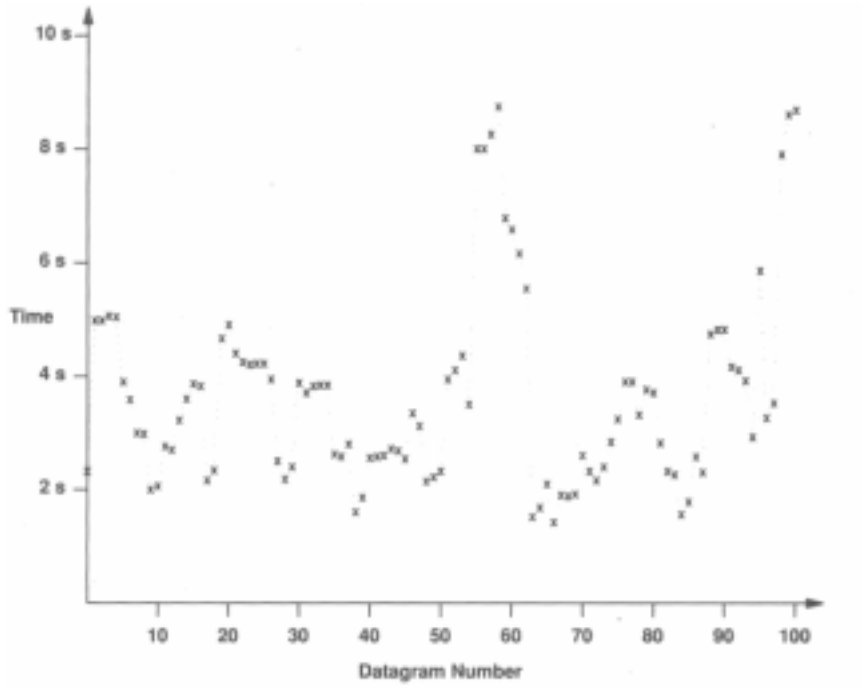

Figure 2. A plot of Internet round trip times as measured for 100 successive IP datagrams. Although most parts of the Internet now operate with much lower delay, delays still vary over time. (Excerpted from [Comer 93]).

4.1 Estimating network latency

Network latency is almost impossible to estimate accurately. The main problem is that round trip times can vary dramatically even from one instant to the next (see Figure 2). This means a good calculation technique has to be able to adapt to a wide variation in delay. We have tried a few different approaches to this problem, and these are described below, along with a discussion on a possibly better solution to the problem.

Ping: As mentioned in section 3, it was not possible to use the original latency measure with the servlet version. Therefore another method had to be found. Since information about the client (such as their IP address) can be found in the HTTP request, using ping seemed like a good idea. Ping is a program used to test the reachability of destinations by sending them an ICMP echo request and waiting for a reply. With Java you can obtain a handle to the runtime environment and execute programs on the underlying system. In this case, the servlet could start a thread that would call the ping program on the departmental web server, with the client's IP

address as a parameter. This approach meant no latency testing code had to be written and seemed a nice tidy solution.

Unfortunately, results obtained from some tests of the ping program (average time after pinging the client 4 times) against time taken to send a set of cases from client to server were not very encouraging. There seemed to be practically no correlation between the average ping time and the time it took to send the cases across. There was also the suggestion that ICMP packets are treated differently by routers and therefore cannot reflect delay, just reachability. Therefore it was decided that ping was not a good measurement tool.

Clock synchronisation: It was mentioned before that the only way to measure one-way delay is if both machines have synchronised clocks. There is much research in this area, and this has resulted in clock synchronisation protocols such as the Network Time Protocol (NTP)[10]. A certain number of computers, equipped with special purpose receivers for time services such as GPS, act as primary time servers and synchronise other computers over a network. These other computers may be secondary servers or ordinary clients.

The synchronisation protocol determines the time offset of the server clock relative to the client clock. The general model followed by all the protocols is described below:

1. Client sends request to server
2. Server sends back a message including its current clock value or *timestamp*
3. Client records its own timestamp upon arrival of the message
4. Since it is impossible to measure one-way delay, RTT is calculated and it is assumed the times are equal in each direction.

Since the model assumes the propagation times are equal, some ways of offsetting errors produced by this assumption must be found. There is a considerable amount of research in this area; however some of the methods require multiple message rounds for each measurement, which would be impractical on the Internet. An important observation is that the correctness of the offset estimate depends on the RTT calculated[11]. Therefore the smaller the RTT, the less influence it has on the offset calculated, and the greater the likelihood of producing an accurate estimate. This is in fact the approach taken by NTP. Several offset/delay samples are calculated and the offset sample associated with the minimum delay is the one chosen as most likely to be accurate.

[10] See http://www.eecis.udel.edu/~ntp/ntp_spool/html/exec.htm

[11] In fact, it can be shown that "the worst-case error in reading a remote server clock cannot exceed one-half the roundtrip delay measured by the client" (see the above URL)

Simple clock synchronisation classes were implemented using the above algorithm. On the client side is a thread that contacts a simple servlet a few times and takes the clock offset to be that calculated during the shortest round trip. The simple servlet merely receives a request and returns its current time in milliseconds.

There were a few problems associated with implementing this algorithm, however. We have to use a servlet as the server-side object so that the firewall problem is not reintroduced. However, this makes the calculation of RTT quite difficult. The problem is, sending a request and receiving a timestamp proceeds like so with servlets:

Create a `URLConnection` object for the servlet
Open connection to the `URLConnection` object
Open an `OutputStream` on the connection
Write data to the `OutputStream` (POST) **(A)**
Open an `InputStream` on the connection **(B)**
Read in data from the `InputStream` (RESPONSE)

This seems fairly straightforward, but there are a number of restrictions. It would seem logical to start timing the RTT at position (A) above, but then the time taken to open the `InputStream` is part of the calculation. The `InputStream` *cannot* be opened before data is written to the `OutputStream`, so there is no way around this. In fact, it turns out that writing data to the `OutputStream` only places it in a buffer and the data is not actually sent until the `InputStream` is opened[12], so timing must start at (B). The `Input-` and `OutputStreams` on the servlet side are also not opened until the client-side `InputStream` is. This means that the estimated RTT ends up larger than the time the object spends in transit. This should be taken into account if the estimation is to be as accurate as possible. The current implementation ignores the above problem and has performed quite well in some informal tests on machines with known offsets. However more rigorous testing and a way that takes the above problem into account are needed.

Assuming the clock offset found is fairly accurate, the next step is to use this knowledge to calculate latency. This is currently being achieved by sending an additional timestamp from the client to the server every time it sends information across. Since the server knows the offset, it is simple to calculate the length of time it took that object to arrive. This is a fairly simple measure, analogous to the original latency metric, and it is possible that it will suffer from the same problems as ping, due to its simplicity.

[12] This is not a well-documented fact and took a lot of searching to find

There is also the additional problem of deciding the ratio between the time returned by this calculation and the time it would take to send X cases across. It may not be correct to assume this a linear function of size, as the size of network packets for example could affect the linearity of the graph. Therefore the ratio should be found experimentally. A way of facilitating this is presented in section 4.4.

TTCP: Perusal of some distributed systems literature (particularly [Schmidt & Gokhale 96]) found mention of a socket benchmarking tool called TTCP, which measures network data transfer rate from machine to machine. However, this was found to be unsuitable for the task at hand - it is used for generating network statistics, and measures throughput from socket open, to read/write of data, to socket close. However, what we require is a way to estimate latency from individual reads/writes.

Discussion

It is possible that the seemingly corrupt results from the ping experiments were in fact due to the high variation in delay which is characteristic of the Internet. High variation was observed between pings immediately following each other, which had seemed impossible, but in hindsight, perhaps they were correct. Another possibility is that estimated latency was fairly accurate, but the delay varied so quickly that it had changed before the cases had finished transferring.

Whether or not this is true of the results found in the experiments with ping, these are things that **must** be considered in any serious attempt at estimating latency. The results are unfortunately indicative of the difficulty of the task. The main thing to be considered when implementing a latency-testing algorithm is that it must be able to adapt to a wide range of variation in delay. Our current and previous approaches take the current delay as indicative of the latency for the next while. As already mentioned, this cannot be assumed, as delay varies quite rapidly. However devising a method, which *does* take note of variance, is not trivial. A possible approach would be to use a similar mechanism to that used by the TCP protocol to calculate round trip timeouts that adapt to a wide range of variation in delay. The 1989 specification of TCP requires implementations to estimate both the average RTT *and* the variance. This information is used to approximate timeouts more accurately, using the following equations[13]:

$$DIFF = SAMPLE - Old\_RTT$$
$$Smoothed\_RTT = Old\_RTT + \delta * DIFF$$
$$DEV = Old\_DEV + \rho (|DIFF| - Old\_DEV)$$
$$Timeout = Smoothed\_RTT + \eta * DEV$$

[13] Taken from [Comer 93]. See also [Jacobson 88]

δ and ρ are fractions between 0 and 1 which weight the contribution the sample makes to the average RTT and the mean deviation, respectively. η is a factor which weights the contribution the deviation makes to the round trip timeout. Research in the area has suggested that values of $\delta = 1/2^3$, $\rho = 1/2^2$ and $\eta = 4$ work well.

Perhaps a similar approach could be used in our system. An estimate of the delay between the server and a client machine is already being calculated every time the client sends a request to the server. However, this calculation could be used each time to update an *average* delay for that client. Therefore each time a client uses the system, the average delay calculation is updated. The mean deviation could also be calculated, and both used in a formula like that above, except with DELAY substituted for RTT (as we are dealing with *one-way* delays) and the following line substituted for line 4:

Latency_estimate = Smoothed_DELAY + η * DEV

This approach would hopefully yield a better estimation of latency. Taking the mean deviation into account should result in latency estimates that are *at least a little* more sensitive to the possibility of change in delay as the cases are in transit.

4.2 Downloading the classes in the background

One important consideration for the feasibility of our approach is that downloading the application logic to the client does not take a prohibitively long time. As mentioned earlier, the ideal solution would be to have the classes download in the background, so that the applet start-up time would not be affected, and to have them downloaded in a compressed form (such as a JAR file), so that the minimum amount of bandwidth is wasted. Unfortunately, background downloads are currently only possible in Java by loading each class individually, which would be too large an overhead. Therefore the entire system is downloaded to the client in a JAR file at the beginning.

Since there is no way of downloading in the background, our aim should be to reduce, as much as possible, the size of the file downloaded at the beginning. A promising way of achieving this aim has been found in a tool called JAX[14], which claims it reduces the size of bytecode files by an estimated 30%-50%. JAX is a Java application packaging tool, written in Java. It works by removing those elements in

[14] See http://www.alphaWorks.ibm.com/tech/JAX

the class files that are not needed to execute the application. The following ordered procedure is followed with each class file[15]:

1.Removal of dead methods and fields
2.Detection of live overridden methods
3.Removal of unused classes and interfaces
4.Inlining of methods
5.Removal of non-essential attributes
6.Shortening of internal method names and field names
7.Removal of non-used entries in the constant pool

A compressed ZIP file is output. Table 1 shows the output produced by JAX when it was run on the classes that need to be transferred to the client.

Analyzed: 17 classes.

Total memory used by jax for processing: 2014424 bytes.

	Zipfile	methods	fields	classes
before	34814	106	124	17
after	20186	64	106	16
savings	14628	42	18	1
	42%	39%	14%	5%

Table 1. The output produced by running JAX on the application classfiles

The savings listed above are very promising, and sending over the zip file produced by JAX instead of a JAR file is an even more efficient way of downloading all of the classes in one step.

An important point worth noting is that no matter which tool is used, JAR or JAX, your browser caches applets, meaning they load extremely quickly the second and on subsequent visits[16]. Assuming your application is not being updated regularly, this means downloading the classes to the client is not such a huge problem after all.

[15] This list is taken from the documentation on the web page
[16] Ironically, this is also a bad thing from the point of view of software updates. Current browsers do not check if an applet has changed on the server before loading from the cache.

4.3 Estimating number of features remaining to be asked

As already discussed in section 3, this is a difficult problem. There could be a research area in trying to determine statistics on correlation of Case Base size + number of features per case + number of possible outcomes, with number of features needed for classification. This wouldn't be easily extendible to Case Bases with continuous-valued outcomes, however. Even if such research were justified (and it might not be), it would probably be too large a task to undertake in the context of this work. There might even be some existing work that would provide us with at least some insight into this, but I am currently unaware of any.

Learning the typical amount of features through use of the system seems to be the best option, however, and requires little effort as the statistics generated from each access can be fed back into the system automatically. Another possibility would be to take the aggressive view that since our aim is to move processing to the client side as soon as possible, in order to increase responsiveness and minimise network usage, perhaps the threshold condition should be

```
(time_to_transfer_cases < 2 seconds)
```

or some similar time constant. Unfortunately, this approach would not work well all of the time, as without a feel for the number of features left to be asked, this approach would result in cases being transferred just as the retriever is close to finding a solution. If this approach were to work at all, the `time_to_transfer_cases` estimate must be as accurate as possible.

4.4 Experimentation

To date, the experiments have not been very successful. Firstly the ping results seemed to show little correlation between observed ping times and case transfer times. This may or may not have been a problem with ping; the problem is, it is difficult to know since we have no other reliable latency measurement to compare it to. It's a catch 22 situation! Testing the effectiveness of our latency calculations is a similar problem. The only metric we can use is that the observed case transfer time varies proportional to our latency estimate. Some small experiments with the current method of estimating latency (the "synchronised clocks" method) have yielded good and bad results. Results on a very slow connection over a modem during peak time showed the expected increase in case transfer time as estimated latency increased. Results from the same experiment over the local network showed no such correlation.

The transfer times all oscillated around a particular point, which might suggest that at such low latencies there is not much variation in package transfer time. The results could also suggest that the latency estimate is simply inaccurate, or that the latency estimate was accurate at the time it was taken, but that the high degree of variation in delay meant this wasn't necessarily representative of the delay conditions as the cases were being transferred.

The main point is – there is no way of knowing whether the bad results were caused by inaccurate latency calculations or not. Therefore a controlled environment, with known latency characteristics, is needed to accurately test our latency calculations. This is also necessary in order to obtain a "transfer index" for sending over cases. Whichever latency calculation we end up using, we need to know the ratio of this estimate to the time it takes to send cases across. The "ideal" ratio could be obtained through experimentation in the controlled environment, and this could be adjusted later to take variation in delay into account. If we attempted to calculate this ratio over the Internet or a LAN, variance in delay would throw our measurements off.

A completely controlled environment would be impossible to achieve, as all communications involve usage of an underlying medium that may experience delay due to load. If signals were sent between two processes on the same computer, the processing needed to both send and receive the data would probably slow down the transfer. If signals were sent over a LAN, delays could be caused by high usage of that LAN. The load on a LAN late at night would most likely be negligible, however sending and receiving on the same computer would probably be the best option.

Filtered Input- and Output Streams could then be implemented. This is simple in Java. The abstract `FilteredOutputStream` class could be extended and a write method implemented which introduced a delay before sending data. Then if our latency calculation produced a value close to this delay, taking delay introduced by the underlying medium into account, we could probably assume the estimate quite accurate. Calculating the transfer index for cases could be performed using this stream also. This stream could also be used to simulate long delays and variance, giving us a controlled network simulator. For example, it could simulate a real network by taking in data on ping times to some remote machine[17], and simulating those delays.

It is hoped that this environment will make it easier to discover the best latency calculation. It might also be useful for later experiments on whether or not our

[17] This data could be gathered overnight by running a constant ping to a machine in America, for example.

approach improves the overall dialog time or not. However an improvement in overall dialog time relies on a good estimate of how long it will take to transfer the cases, as if they are sent over when the shouldn't be or vice versa, the system will not perform very well in those experiments. Sending over cases unnecessarily, and at too high a transfer time, would increase dialog time. *Not* sending them over when they should be transferred would result in a long-lived dialog over the network that could suffer from unresponsiveness. Therefore our first priority is to find a good latency calculation.

5 Conclusions and further work

5.1 Further work

There is still much work to be done on the current strand of this project. Once we are happy with our method of determining the best time for sending the cases across, we must then perform experiments to determine if this load-balancing approach is advantageous to the user. It is reasonable to assume that response time will improve when processing is taking place on the client's own machine, but we need to determine if this improvement in time is larger than the extra time needed for the migration of code and data. These experiments will be conducted in the immediate future.

For the experimental stage, we will need to find more Case Bases (or construct synthetic Case Bases), which reflect the benefits of load balancing. It would be expected that the ideal Case Bases would be those that have many features and possible outcomes, as a greater amount of dialog would be required to reach a solution. This will be verified through experimentation on Case Bases with various characteristics.

The broader area of this research is in decentralised retrieval models over networks; therefore in the long-term we hope to investigate other possible load balancing models for CBR systems. Some possible models we could explore are discussed briefly below.

One possibility would be a truly distributed system, where the retriever itself is distributed – many machines working on retrieving the best case using distributed Case Bases (perhaps different subsets, like a parallel retrieval system). If the entire Case Base is distributed however, the problem of keeping it consistent across different

machines must be addressed. With a pseudo-parallel retrieval system, there is also the problem of integrating the results found by different machines. This kind of system could not work with incremental retrieval, as each refining question would have to be sent to multiple servers, increasing network traffic. It could be an efficient method of "one-shot" retrieval however, as solutions would be found quicker due to the parallel processing. All of the solutions found could be sent to the client for integration, which would balance the load nicely between machines.

Another approach would be a distributed model that uses the footprint-based method of retrieval [Smyth & McKenna 99]. This two-stage retrieval process offers a clear-cut point at which processing could be transferred. The server could retrieve the "footprint" case, and send its related set over to the client for further processing. However it seems that the related set might be quite large, so sending it to the client at runtime might not make sense due to the network load created. And as the appropriate related set is naturally not known until the footprint case has been retrieved, there is no way of "prefetching" the related set to speed up the process. Therefore, although there is a distinct advantage in distributing this retrieval process from the point of view of server load, sending over processing on the fly would not be practical due to the load created on the network.

A better way of distributing this retrieval process would be to distribute the "footprint sets" themselves (i.e. place local copies on certain client machines that query the system on a regular basis). In this scenario, retrieval of the footprint case would be performed on the client machine. The server would be contacted with the footprint case, and would have the task of finding a suitable case in the footprint case's related set. This distributed retrieval process would result in reduced load on the server rather than the network. However, since client machines are performing the initial retrieval stage themselves, queries to the server may be a little more spaced, reducing congestion in the vicinity of the server. This model would also be of use in an Intranet system, where many client machines make use of a central system throughout the day. An example would be an internal helpdesk system or machine translation system[18]. Although an Intranet server would not be as overloaded as a web server would, it should still display increased output due to the distribution of processing load.

[18] This could be used for localisation tasks. We are considering implementing a distributed Example-Based Machine Translation system as part of this work. The distribution would be performed by distributing the "footprint sets" as described above, and the EBMT part would be based on the work of Collins [Collins & Cunningham 96]. Attempting to apply the theory behind footprint-based retrieval to the EBMT cases, to see if were possible to discover "footprint sets" and "related sets" in this domain, could yield interesting results.

Another possible way of reducing server load and network latency would be to have a geographical push-caching system such as that proposed by Gwertzman (see [Gwertzman & Seltzer 95] and section 2). Instead of pushing documents, this system would push Case Bases and application logic to less loaded servers at off-peak times. Therefore network latency would be reduced, as clients could contact their nearest CBR server. However this kind of system is not really within the scope of this research.

5.2 Conclusions

This paper presented a method of load balancing in an intelligent client-server system that involves dialog. There are a few reasons for wishing to move away from the traditional client-server model, where all of the processing takes place on the server. Moving the computing load to the client allows the server to serve larger groups of users with only modest resources, and also reduces the cost for large systems. Network traffic is also reduced on both the client's local link and in the vicinity of the server. Internet traffic has grown at such a rate that high delays are common and so systems that send data back and forth suffer from unresponsiveness. Therefore in a situation where there is a long-lived dialog, a good case can be made for sending processing over to the client.

In section 1, a brief introduction to Case-Based Reasoning and I-CBR was given. Our reasons for wishing to adapt the traditional client-server model were explained in section 2, which discussed current network problems on the Internet. The details of how the current system was developed, stage by stage, were described in section 3, and the main research issues emerging from the current work were discussed in section 4.

We feel there is a good argument for client-side processing, but the load-balancing method we describe would only be practical in the context of a system involving dialog. We must ensure that the cost of migration of processing does not outweigh the potential response-time benefits. Ensuring this is only possible when the time cost of transferring the Case Base can be measured against the time that would have been spent on sending queries and responses back and forth. There could be no benefit to sending over processing at runtime in a one-shot retrieval situation.

In the future we hope to look at other ways of balancing client/server load in network-based retrieval systems, with a possible application in the area of distributed machine translation.

6 References

Aamodt A. and Plaza E. (1994), *Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches.* AI Communications, 7(i), pp. 39-59.

Collins B. and Cunningham P. (1996), *Adaptation-Guided Retrieval in EBMT: A Case-Based Approach to Machine Translation.* In Proceedings of EWCBR'96, Advances in Case-Based Reasoning, Lecture Notes in Artificial Intelligence, Smith I. and Faltings B. eds., Springer Verlag, pp. 91-104.

Comer D.E. (1995), *Internetworking with TCP/IP, Vol I: Principles, Protocols, and Architecture*. 3rd ed., Prentice Hall International.

Cunningham P. and Smyth B. (1994), *A Comparison of Model-Based and Incremental Case-Based Approaches to Electronic Fault Diagnosis.* AAAI '94 Workshop on CBR (Seattle), Aha D. ed.

Doyle M. (1997), *Web-based CBR in Java*. Final year undergraduate project, Department of Computer Science, Trinity College, Dublin.

Ferguson P. & Huston G. (1998), *Quality of Service on the Internet: Fact, Fiction, or Compromise?* INET '98

Gwertzman J. & Seltzer M (1995), *The Case for Geographical Pushcaching*. In Proceedings of the 1995 Workshop on Hot Operating Systems.

Hine J.H., Wills C.E., Martel A. & Sommers J. (1998), *Combining Client Knowledge and Resource Dependencies for Improved World Wide Web Performance*. INET '98

Jacobson, V. (1988) *Congestion avoidance and control*. In Proceedings of the ACM SIGCOMM '88.

Labovitz C., Malan G.R. & Jahanian, F. (1997) *Internet Routing Instability*. In Proceedings of the ACM SIGCOMM '97.

Markatos E .P. & Chronaki C.E. (1998), *A Top 10 Approach for Prefetching the Web.* INET '98

Quinlan J.R. (1986), *Induction of Decision Trees*. Machine Learning, Vol. 1, No. 1, pp. 81-106.

Quinlan J.R. (1993), *C4.5: Programs for Machine Learning*. Morgan Kaufmann.

Riesbeck C. and Schank R. (1989), *Inside Case-based Reasoning*. Lawrence Erlbaum.

Schmidt, D. and Gokhale A. (1996), *Measuring the Performance of Communication Middleware On High-Speed Networks*. ACM SIGCOMM 96 Conference (Aug. 96, Stanford University).

Smyth B. and Cunningham P. (1995), *A Comparison of Incremental Case-Based Reasoning and Inductive Learning.* In Advances in Case-Based Reasoning, Lecture Notes in Artificial Intelligence, Haton J-P, Keane M., and Manago M. eds., Springer Verlag, pp.151-164.

Smyth B. & McKenna E. (1999), *Footprint-Based Retrieval*. To be presented at ICCBR '99.

Willcox S. (1997), *Designing Thin Java Client Applications for Network Computers*. At URL: http://java.sun.com:81/javareel/isv/Avitek/ThinClientWP.htm.