ESPRIT Project No. 25 338

Work package G

Service Deployment

Final Report

ID:	DG6.2	Date:	01.03.99
Author(s):	**INRIA**	Status:	
Reviewer(s):		Distribution:	

Change History

Document Code	Change Description	Author	Date
DG6.2	First version of document.	INRIA	01.Mar.99

1 Introduction

The work accomplished in the context of the Service Deployment workpackage applies both to ETEL and ETEL++. Both took advantage of the use of the Monitoring tools that were released during the summer 1998. In the case of ETEL, these monitoring tools made possible the use of prefetching strategies that increase the hit ratio in the data buffers of users. Prefetching tries to predict what pages could be accessed next, and fetches the identified pages lazily in the background. This enhances the performance of the client side of the system. In addition, we built on Monitoring tools and investigated the use of Data-Mining and clustering techniques applied to clusters of computers to enhance the performance of the server side of the system.

Although ETEL++ delivers electronic editions to users, its architecture is radically different from the one used for ETEL, and therefore, some of the ideas that maturated during our studies were more delicate to apply to this context. ETEL++, however, offers a wide spectrum of Service Deployment related problems, and we therefore focused our studies on the ones that are raised by the global network environment that link together ETEL++ machines. In this context, the monitoring tools made possible the design and implementation of a load balancing strategy that dynamically determines the route data-flows have to follow to minimize response times.

This paper is a summary of the work achieved in the context of the Service Deployment workpackage. It therefore presents the monitoring tools. It then gives a typical example of the way these tools should be used by an application. The next section is a summary of our experience using the tools we developed. It then details the Data-Mining and clustering approaches and presents in addition the load balancing strategy that runs inside ETEL++. Before presenting the above technical aspects, we detail why lowering the response times of the accesses to electronic editions is a major constraint.

2 Timeliness and Scalability

The primary concern for the design of both ETEL and ETEL++ is to guarantee the scalability of the electronic newspaper service. In general, the scalability of a system determines its ability to offer a collective service without degrading the quality of the service for individuals. Therefore, electronic editions must be made available to a growing number of users without increasing the response time of the interactions with each individual user.

For both systems, we tried to enforce the following property: the time taken to read an electronic newspaper edition must be close to the one that would be taken to read its counterpart paper edition. Ideally, this means that a requested page of an electronic edition has to be made available to a user in a time frame that is close to the typical time taken to consult a particular page of the paper-based edition. This property can be translated in two constraints:

- *Timeliness* at the level of each user, i.e., providing fast response time to every user.
- *Scalability* of the server, i.e., supporting a large number of users at the lowest cost in terms of resource needs, while maintaining timeliness.

One possible way to enforce these constraints is to have the electronic newspaper locally available on the machine of each individual user before he accesses it. This solution, however, is not realistic, partly because of the resulting storage costs. One Ouest-France daily edition may be made of about 400 pages, and each page has an average size close to 80 Kbytes.

These two constraints had a significant impact of the architecture of both ETEL and ETEL++. ETEL and ETEL++ differ, however. The architecture of ETEL is somehow classical, and can be illustrated by Figure 1. Its architecture relies on the client-server model. The clients are the users, which connect to the system via an ISDN dedicated network. The server is made of one or more clusters of several machines, interconnected by fast links.

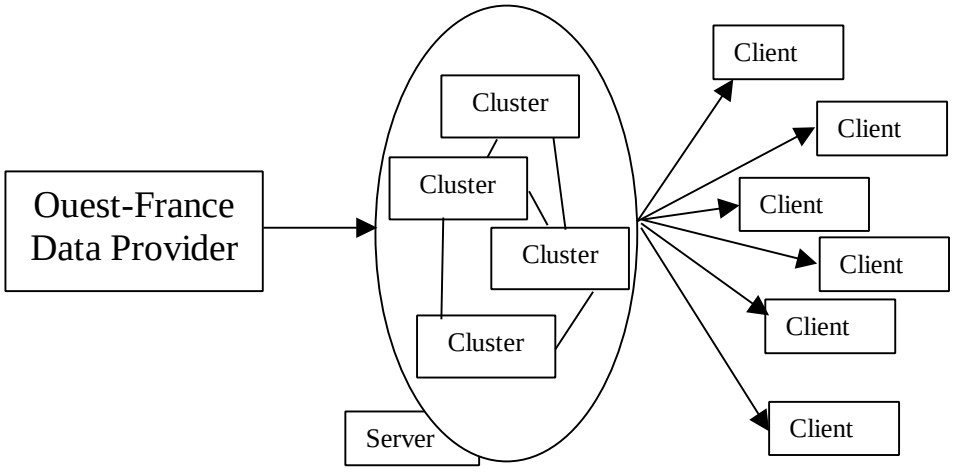

Figure 1: The Architecture of ETEL

Basing the server on clusters of machines was a major design decision. This decision was motivated by the use of Data-Mining algorithms together with clustering algorithms. This is detailed in Section 5.

The architecture of ETEL++ is different because it does not rely on a typical client-server model, but on a model in which any machine involved in the tasks of producing editions can be a server, a client or both (this type of architecture is sometimes called *peer-to-peer*). The distribution of information and processing tasks is a predominant factor. That architecture is described in details in the document DJ4, and depicted by the Figure 2.

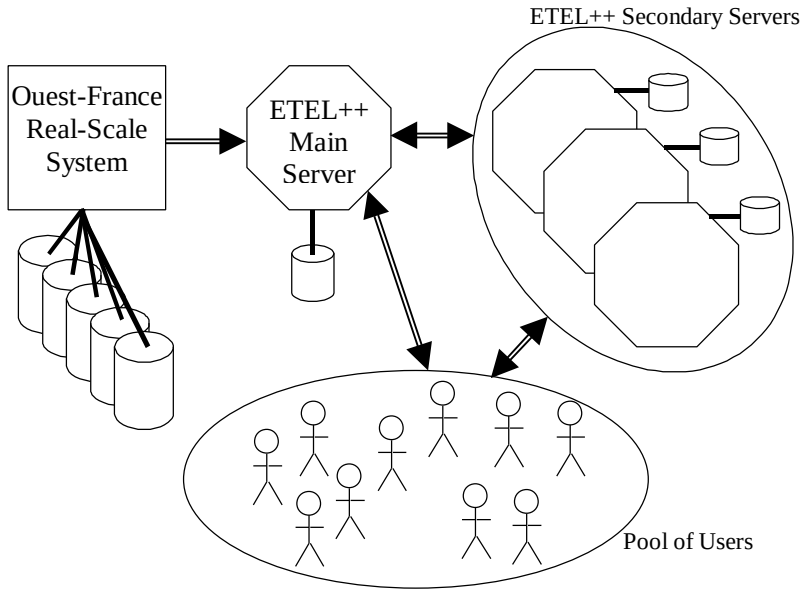

Figure 2: The Architecture of ETEL++

We conducted two investigations to enforce the above constraints. First, we investigated the need for monitoring tools to analyze the performance behavior of systems. We also investigated the need for Data-Mining techniques used together with Clustering techniques. In parallel with these investigations, we worked on the design of load-balancing policies that would make possible the enforcement of the constraints specified above.

The outcome of these investigations is presented in this paper, and can be summarized as follows:

- It is necessary to provide applications with handy ways of monitoring the usage of arbitrary resources, as this monitoring is at the very root of any deployment.
- Data-Mining used together with Clustering proved to enhance the performance of classical client-server architecture, therefore enforcing the scalability of the whole system.
- In the case of peer-to-peer architectures, performance sensitive routing of data-flows tends to minimize response times and to limit the occurrences of bottlenecks.

Section 3 presents the monitoring tools that have been released. Section 4 presents a roadmap that should help developers in setting a monitoring environment. Section 5 presents both the Data-Mining and the Clustering approaches that have been implemented and evaluated within the context of ETEL. Section 6 presents the load-balancing strategy that is used within ETEL++.

One major result of the work achieved in the context of this workpackage is that is has been possible to build very complex deployment policies on top of the observation framework developed, as we did for ETEL and ETEL++. The extensive use of that framework demonstrated that is was rather convenient to use, and that is was possible to define arbitrary new resources to monitor (see Section 3). The (almost only) difficult problem is to implement the hardware (or platform) specific code that interacts with real resources (CPU, …) and that uses the appropriate system-calls the right way. The lowest levels of the tools need to go down to this level of details, which is not always simple, especially when the documentation describing these calls is vague or simply missing.

3 Monitoring Tools

Monitoring Tools have been designed to give applications convenient ways to extract useful information from the monitoring of arbitrary resources, and in turn to take performance-related decisions. In a nutshell, monitoring tools are *mechanisms* for evaluating the usage of resources. Application may use these tools to enforce *policies* achieving a particular goal, load balancing being a possibility. This separation between mechanisms (that are part of this work package) and the policies (that are part of other work packages) is important. Therefore, the tools described here have to be generic enough for being possibly used in different contexts and for being included in various policies.

Document DG3 presented the design of these tools. We therefore briefly restate the main concepts detailed in that document.

So to speak, the ultimate goal of monitoring tools is to be part of a performance-aware policy. It is therefore a requirement to provide applications with rather general-purpose tools that can monitor any possible type of resource. The design of the tools decouples the general, somehow abstract, notion of a resource from any concrete, low-level hardware component. A resource is viewed as a black box that, on request, returns values representing its usage. Therefore, a resource can typically be associated to a CPU, or be an application-specific notion like for example the number of users that connected to a service on a given date.

Once the resources to monitor have been defined, it is necessary to define the type of monitoring to apply on each resource. Monitoring can be periodical or aperiodical. For periodical monitoring, it is necessary to set the frequency of the monitoring. Aperiodical monitoring can either be request-based or notification-based. A request-based aperiodical monitoring returns the current value of a resource usage only on-demand. A notification-based aperiodical monitoring returns values only when the current usage is above, and/or below, a threshold set by the application.

Usage values that correspond to the usage of resources are kept in specific data structures (called Histories) that can be interrogated by applications. On the one hand, the performance monitor query the resources and produce values inserted in these structures. On the other hand, applications consume some of the inserted values. This producer/consumer mechanism enable to implement arbitrary filters that may transform the raw produced data into sophisticated

consumed values. For example, it is possible to apply high-gain filters, or aggregate operators (like average) to the values consumed in order to get a high-level view of the underlying usage of the resources.

Performance monitors are globally accessible, that is, the monitoring each performs can be queried by applications running on the same host, or by remote applications by the means of the trader provided by APM. It is therefore possible to obtain a global view on the performance of a distributed system before enforcing a performance-related decision. This is actually accomplished by ETEL++.

To summarize, the monitoring tools have two aspects. The first aspect is querying hardware resources, to obtain for example the load of the CPU, the bandwidth of network links or the amount of memory. The second aspect is defining the way each resource has to be monitored (defining the resource, the type and frequency of observations, etc.), and making available the observed values in a multiple sort of ways. This last part has often been referred to as the observation framework of the Service Deployment.

The observation framework, however, completed last summer, is fully operational, as detailed in the document DG3. The first part, querying the hardware resources, is not complete mainly because of the complexity and lack of clarity of the interfaces available on Windows platforms (the ones we currently use for development). Today, the detection and the observation of values is performed by the analysis of configuration files, and not however by querying real hardware resources. The core of the Service Deployment is ready, that is, tools for performing periodic observation and for enabling general definition of what is a resource and how to observe it (indeed, the configuration files we use *are* a resource). This missing part consists of the lowest levels of the Service Deployment work. As indicated in the Document DJ4, the validity of what we already developed is demonstrated by the policy that actively runs inside the Pilot application and that drives the flows of data.

4 The Roadmap Applications Should Follow to Use the Monitoring Tools

This section presents a typical example of the declaration of a monitoring tool, and also the way the declared monitor can be utilized by an application. This example can be considered as a template for declaring any other type of monitoring for different resources. The first part of the example is the declaration of the monitor. The second part gives sample code for using it. The complete API of the Monitoring Tools is given in the annex at the end of this document.

4.1 Declaring a Performance Monitor and a Monitored Resource

The following code creates a performance monitor and creates a resource that is monitored. The performance monitor is created within a Place (according to APM terms). This monitor is in charge of managing all the monitoring tasks on the machine hosting the place. Therefore, any other task that wants a resource to be monitored has to get a reference on this performance monitor via the TrivTrader. In the case of the code given below, the monitor and the resource are created within the same Java task, and thus the reference to the monitor is known.

Once the monitor created, a resource is defined and made available to the monitor for periodic observation. In the case of this example, the period is 4 seconds. A simple measurement is defined (a *BasicMeasurement* is allocated), this means no specific processing of the values returned by the resource (the CPU) is enforces at request time (every 4 sec). Note that the observed values are accumulated in a history that is declared.

```
import fr.inria.ServiceDeployment.*;
import fr.inria.Util.*;

public class MyPlace extends MonitoredPlaceImpl implements MyPlace_int  {
      // Variables for monitoring.
      // We show an example with monitoring solely a CPU.
      // We almost always need 4 such variables per monitored resource.
      Target targetCPU = null;
      Measurement measureCPU = null;
      History historyCPU = null;
      CPU resourceCPU = null;

      // Declaration of the Monitor on the current Host that
      // watches over MyPlace.
      MonitorImpl theHostMonitor = null;

      public setupMonitor() {
            //----------------------------
            // Action 1.
            // allocate the monitor and register in the TrivTrader
            // (assumed already known).
            theHostMonitor = new MonitorImpl(trivTrader);
            trader.put("myPlace", theHostMonitor, Monitor.class);

            //----------------------------
            // Action 2.
            // Register each resource that should be monitored.
            // Action 2.1: allocate the relevant resource and publish it.
            // the class CPU provides a method getValue().
            // Any other resource must do the same.
            resourceCPU = new CPU();
            addResource("resourceCPU", (Resource)resourceCPU);
            // Action 2.2: define the target of the measurement (here
            // the place) and publish it.
            targetCPU = new Target("myPlace");
            // Action 2.3: define the type of measurement and the observation
            // frequency (in this case; other observation modes
            // presented in DG3).
            measureCPU = new BasicMeasurement("resourceCPU", targetCPU);
            historyCPU = new History(theHostMonitor,
                                     measureCPU, new Frequency(4000));
      }
/*
      the rest of the class, including some code to get the TrivTrader
*/
}
```

4.2 Accessing a Performance Monitor

The second example of code shows how to access the monitor previously defined. A simple way to do this is to create a method in the class MyPlace that returns the usage of a resource. It is

therefore recommended to create an interface to MyPlace, to register this class in the trivtrader, and to (possibly remotely) access this interface to get the desired value. A possible implementation can be:

```
public int getMips(){
      int r = -1;
      try {
          r = ((IntValue)theHostMonitor.getEvaluation(measureCPU)).getValue();
      }
      catch (Exception e){ /* do usual exception handling */ };
      return r;
    };
```

4.3 Experience with ETEL++

A similar scheme has been used in ETEL++ to obtain the values reflecting the workload of each server in order to determine an optimal data-flow map. In the case of ETEL++, two resources are declared: a CPU (as in the example) and a network. The observation mode of these resources has been set to the request-based mode, as in the above example. The period of observation has been set to 1 second (this is typical of traditional monitoring tools such as the Unix xload). The measurement returns a percentage of usage.

These two resources can be queried across the network, making therefore possible for any remote site to determine the current (CPU, Network) load of any other site. In the case of the deployment policy, as detailed Section 6, the main ETEL++ server dialogs with all other secondary servers to get their workload, and then computes the roadmap.

5 Data-Mining and Clustering[1]

The ETEL server is structured as a set of autonomous clusters of machines, each having a significant secondary storage space and a powerful processing unit, and that are interconnected by a high speed network. This design choice was motivated by various factors:

- Clustering is recognized as a useful technique for improving scalability when applied to either data elements or processing elements.
- The configuration of a system may easily be changed so as to face new environmental factors. In particular, new processing elements may be added at low cost when the number of system users is increasing significantly.
- The computing infrastructure of the Ouest-France editor is geographically distributed over a number of sites, each site producing distinct local news. Hence, each of these sites may be used to host a cluster.

Clustering of processing elements is not sufficient to address the scalability constraints. We must further deal with the distribution of the newspaper data and of the client requests over the clusters. Base distributed system techniques are eligible here: data replication and load balancing. Data replication consists of having multiple copies of a data over distinct processing elements. Concurrent accesses to a data may then be scheduled over the various processing elements storing the data, with respect to load balancing and access latency considerations. The drawback of data replication is that it requires managing replica consistency. However, there is no need for consistency management in the system. Clients perform only read accesses. Thus, consistency issue raises only when a new edition is produced, i.e. when the content of the new edition is distributed over the various clusters. Clusters being autonomous, a user interacts only with one cluster, which renders useless inter-cluster consistency. Finally, regarding intra-cluster consistency, a new edition is produced at night, and thus at off-peak hours. It follows that we have decided to not deal with consistency management in the system since it has a negligible impact on the overall quality of service of the newspaper.

[1] Please refer to the Document DG3 in which many references to the state of the art in Data Mining are given.

Although data replication and load balancing are *a priori* eligible techniques for improving scalability, they must be used in an appropriate way to be actually beneficial. For instance, data replication must not lead to overload the clusters' storing elements. In the same way, the load balancing strategy should not introduce processing overhead when processing elements are handling requests. Our approach to offer scalability guarantees using clustering, replication, and load balancing techniques relies on exploiting the user profiles in terms of accessed topics. Knowing the topics that are the most frequently accessed by each user, we are able to identify groups of clients such that the clients of a group read similar pages. In the same way, we are able to identify groups of pages such that the pages of a group are all accessed by some set of clients. Groups of pages constitute units of load balancing for data distribution over clusters while groups of clients constitute units of load balancing for request distribution over clusters. Achieving a high degree of scalability for the server then lies in the adequate computation of user and page groups. The next subsection introduces our technique for computing these groups, and the associated load balancing strategy.

5.1 Profile-based load balancing

Our objective is to compute groups of clients and associated page groups given user profiles. The grouping strategy should be such that it enhances scalability of the cluster-based server. In other words, we must compute groups so as to be able:

- To map page groups onto clusters, according to the clusters' storage capacities, in a balanced way, while ensuring that the resulting distribution of user requests among clusters will also be balanced.
- Upon the initiation of a user session, to assign the corresponding client requests to a cluster, according to the cluster's load and the page group stored by the cluster.
- To guarantee fast response times to user requests, i.e., ensuring local availability of the requested page within the cluster.

The aforementioned requirements may be achieved by providing a grouping strategy computing a hierarchy of page groups that is such that:

- Every page group is accessed by a *significant* number of users, i.e., a page group belongs to a significant number of user profiles. This enables the system to enforce a balanced processing load among the clusters.
- Within the hierarchy, page groups are related in terms of common pages, which enables to identify a number of page groups equal to the one of clusters without sacrificing the resulting load balance.

Remark that a page may belong to various computed page groups, which determines the replication of pages based on their popularity among users. In addition, a single user profile may include distinct page groups, which provides the adequate framework to deal with processing load balancing among the server's clusters. Other requirements for the grouping strategy is that

it should have minimal spatial and temporal complexities, and should be dynamic so as to efficiently handle changes at the level of either the users base or the server configuration.

Existing algorithms for computing client and/or page groups relate to the data analysis and data mining domains. The following paragraph presents a novel algorithm that we proposed based on the result of this evaluation, the algorithm being optimal with respect to the aforementioned requirements for the grouping strategy.

5.2 Grouping algorithm

Our algorithm is based on associative data mining. Briefly stated, associative data mining computes a set of inference rules among database elements (or *items*), according to the transactions stored in the database where each transaction contains a set of database elements (or *itemset*). In our context, an item corresponds to a newspaper topic and a transaction corresponds to a user profile. An inference rule of the form *I -> J* means that most transactions containing the itemset *I* also contain the itemset *J*. In general, an inference rule is associated with its *support*, which gives the number of transactions verifying the rule, and its *confidence*, which gives the probability with which a transaction containing *I* will also contain *J*. A classical data-mining algorithm decomposes in two phases:

- In the first phase, the algorithm computes the sets of elements that are frequent within transactions. Frequency is determined according to a given support threshold. This process is done iteratively by computing at the *k-th* step, the frequent sets containing exactly *k* elements. The process terminates when a frequent set cannot be identified for the given number of elements.
- The second phase consists of identifying inference rules given the frequent sets computed in the first phase. Basically, an inference rule of the form *I->J* is identified if the number of frequent sets verifying this rule is greater than a given threshold.

In our context, the above first phase of the algorithm computes page groups of increasing size, and each group identifies a set of topics that are all accessed by a number of users exceeding the given threshold. In other words, it gives us with groups of topics that are significant (or frequent) enough to be used as a basis for the distribution of pages and of user requests over the server's clusters. However, existing algorithms return a set of independent group. In order to get a hierarchy of groups, we modify the algorithm's first phase so that it structures computed groups according to a tree structure. The modification consists of establishing a hierarchical relation among (frequent) groups containing *k* topics and those containing *k+1* topics. The unique ancestor of a group containing *k+1* topics among eligible groups of *k* topics is selected according to an ordering among topics: the additional topic of a descendant is greater than all the topics of its ancestor. The ordering among topics is set arbitrarily; for instance, it can be the alphabetical order over topic names. Given the aforementioned modification, we are able to compute a hierarchy of groups for an algorithmic complexity of *O(N)*, *N* being the number of registered users. However, the data-mining algorithm is not suited for the dynamic computation of groups. Any modification within the set of user profiles leads to re-compute the whole hierarchy. One

way to deal with this issue consists of periodically executing the algorithm. Such a solution is suitable only if the modifications of profiles occurring between two algorithm executions do not affect significantly the current grouping. We cannot make this assumption, which has led us to propose a second adaptation of associative data mining so as to make the algorithm dynamic.

Our algorithm dynamically builds the group hierarchy using recursive procedures for the addition, removal, and modification of a user profile. By construction, a tree node is necessarily a frequent group whose level in the tree determines its size. Let us consider the introduction of a user profile that includes the topics of a frequent group whose corresponding node is at level *k*. Depending on the topics given in the user profile, this may lead to identify a frequent group of *k+1* topics, and hence to create a descendant--if it does not exist already. Thus, each node of the tree stores the number of clients whose profile contains the topics of its associated group and an additional topic (which has to be greater than all the topics of the group). However, in order to minimize the memory space used by the algorithm, this information is not kept for all the additional topics. The number of clients gets counted once the corresponding group can potentially become frequent. The frequency potential is evaluated with respect to the data mining principle: a frequent group of size *k+1* is necessarily composed of frequent groups of size *k*. Hence, a group of size *k+1* can be considered as being potentially frequent only once all the groups of size *k* that it contains are themselves frequent.

5.3 Load-Balancing Strategy

Given the hierarchy of groups computed using our dynamic data-mining algorithm, the load balancing strategy decomposes into:

- The distribution of an edition (i.e. the electronic version of the 40 Ouest-France daily regional editions) on the server's clusters, upon the edition's publication, which happens at night. Distribution of an edition could also have to be initiated when the hierarchy of groups changes or when one of the clusters fails. We do not consider these cases because their frequencies of occurrence are less than the one of edition production.
- The distribution of the users' requests upon users connections, on the server's clusters.

Let us first detail distribution of the edition. The objective here is to map page groups onto clusters so as to avoid the overloading of clusters in terms of processing and storage usage. This is achieved by distributing the leaves of the group hierarchy over the clusters. Since the number of leaves in general exceeds the one of clusters, page groups of the leaves are grouped based on ancestor commonality, using the group hierarchy. In addition, so as to optimize the placement of page groups with respect to the resulting processing and storing load, the grouping of inner page groups is done with respect to the distance among profiles of associated users. Notice that the mapping of groups onto clusters need only to be computed when there is a modification of the group hierarchy that is such that the mapped groups are no longer valid. Given the mapping of page groups onto clusters, the pages of the edition are transferred to clusters accordingly. For the distribution of pages among the machines of a cluster, this is realized by distributing the pages

over each machine in turn. This simplistic choice is due to the fact that an efficient cooperation protocol is implemented among the cluster's machines. Efficiency comes from the use of an *ATM* network but also from the cooperation algorithm itself, which is detailed in the next subsection.

Concerning distribution of the users' requests, it consists of choosing a cluster for every user upon the user's connection. The cluster is selected based on the groups it hosts and on its current processing load. Among clusters that host a group, which is contained in the user's profile, the one whose associated group is the largest, is selected in priority. However, if its current load does not enable it to host the client, remaining clusters are considered, still based on the size of the groups they host. When more than one cluster is eligible, the least loaded cluster is selected. Up to this point, we have given the selection of clusters upon client connections. However, a client interacts with one machine of the cluster, this machine will be the least loaded one among the cluster's machines.

5.4 Performance Evaluation

For performance evaluation of the server from the standpoint of scalability, we have to run experiment with a large number of clients (up to 100.000) accessing various large server configurations (up to 10 clusters of 10 machines). Since we were not able to have a prototype running with the large system configurations that had to be considered, this led us to use simulation for performance evaluation of the server.

5.4.1 Simulation environment

The simulation environment we developed for the server is implemented in the C++ language. Temporal aspects, i.e. delays among client and server interactions, are dealt with using the CSIM simulation library. The system architecture is depicted Figure 3; it is composed of the 6 following modules:

- *Generator of client profiles*: The generator module handles the creation of client profiles, based on a sample survey about readers habits made available to us by Ouest-France. A client profile embeds the time at which the client starts accessing the service, the set of accessed pages together with reading time for each page.
- *Data Producer*: The Data Producer is roughly simulated and includes three operations for respectively reading, writing and removing an element in the database.
- *Grouping module*: The grouping module computes page groups according to the dynamic data-mining algorithm.
- *Client module*: The client module serves two purposes: (i) it manages the clients base by handling the creation and removal of clients, (ii) it manages client requests by simulating the user sessions for all the clients. Both actions are carried out using the base of user profiles. In addition, two implementations of request management are provided: one corresponds to the profile-based prefetching strategy, and the other is a simple fetch-on-request strategy.

- *Load balancing module*: The load balancing module implements our load balancing strategy, called PROFILE, presented above; it selects the server's machine upon the connection of each client based on the group hierarchy computed by the grouping module and the machines' load. In addition, for evaluation of the ETEL server, we implemented two alternative load balancing strategies: one strategy, called STAT, is simple and consists of statically assigning a machine to the client based on the client identity; the other strategy, called DYN, is dynamic and consists of selecting a machine based on the machines' load. Let us remark here that when using our load balancing strategy, the client module runs the profile-based prefetching strategy, while the client module runs the fetch-on-request strategy when running either the STAT or DYN algorithms.
- *Server module*: The server module simulates the server architecture, including cluster management. In addition, regarding evaluation of the ETEL server, we implemented another cluster management strategy. This strategy uses a simple multicast protocol for cooperation, and is run when executing either the STAT or DYN load balancing algorithms within the server module.

Given the above modules, the simulator can be decomposed in two distinct processes, denoted in Figure 3 by plain and gray arrows, respectively.

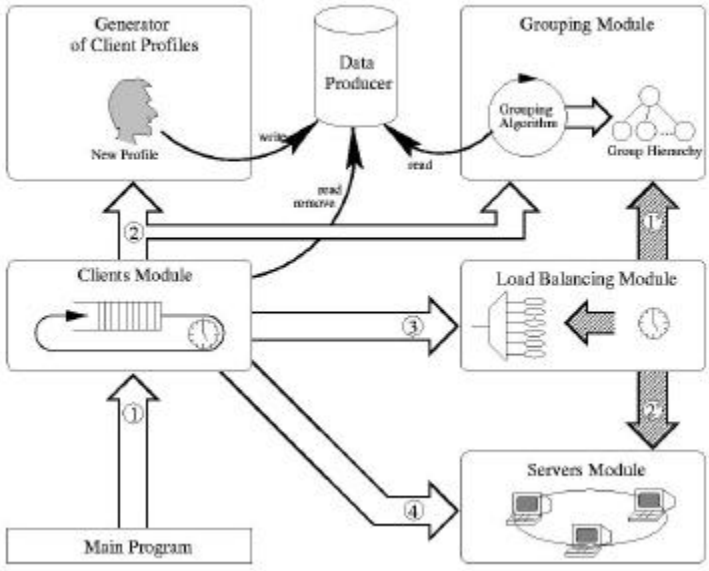

Figure 3: The Simulation environment

The first simulation process relates to the simulation of client requests. The main program creates the client module (arrow #1) whose role is to dynamically create, modify and remove clients as well as to simulate their requests. The creation of a client (arrow #2) leads to call the generator of client profiles that stores the resulting profile in the database, and to call the grouping module, which dynamically adds the profile in the existing group hierarchy. User sessions are also simulated within the client module. In a first step, the client contacts the load

balancing module, which notifies the client about the machine it must interact with, based on its knowledge of the group hierarchy and of the machines' load (arrow #3). Then, the client directly interacts with the selected machine (arrow #4), which is managed by the server module.

The second simulation process relates to the daily production of the electronic edition. At a fixed time of the day, the load-balancing module reorganizes distribution of page and client groups over the server's clusters. Towards that goal, the module first gets the up-to-date group hierarchy (arrow #1'), uses it to infer the corresponding distribution of data groups over the clusters and then to distribute the data accordingly (arrow #2').

5.4.2 Performance Results

In the performance evaluation of the server, our goal is to show the ability of the server to manage a large number of users without degrading the system's response time from the perspective of each user. Such ability of the server is thus characterized with respect to the number of clients and to the mean response time offered for client requests. The system's response time to a client request depends on three factors: (i) the time taken to transfer the request from the client to the client's server machine, (ii) the time taken by the server for handling the request, and (iii) the time taken to transfer the requested page. In the system, the interaction between a client machine and a server machine is achieved using a persistent ISDN connection. Thus, the time taken for transferring a request and corresponding result is fixed for a given message size, and is independent of the time taken for establishing a connection and of the network load. It follows that we measure only the time taken for handling requests in the following.

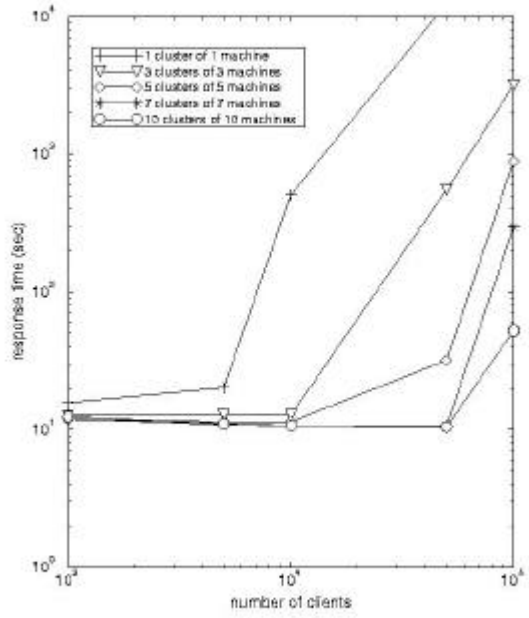

Figure 4: Performance of the STAT Algorithm

Figure 4, Figure 5 and Figure 6 give the server's mean response time with respect to the number of clients, for the three simulated load balancing algorithms (STAT, DYN, and PROFILE). For each algorithm, we ran simulations for various server configurations, which differ according to the number of clusters and to the number of machines per cluster. Notice that the configuration with one cluster of one machine corresponds to a centralized server and is given to show the behavioral equivalence of the three simulations. For remaining configurations, results show that our algorithm performs better than the two others do.

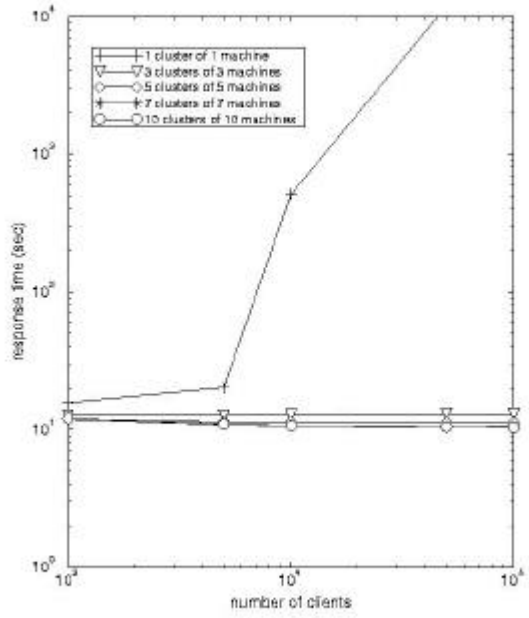

Figure 5: Performance of the DYN Algorithm

Given the curves of the figure, one can see that the time taken for handling a request for a fixed number of clients decreases as the configuration increases. This is a straightforward result since requests are distributed over the server's machines and hence the larger is the configuration the less is the load of each machine. Let us now concentrate on the average response time observed for the STAT algorithm, for a given server configuration. One can identify two phases in the curve: (i) for a small number of clients, the average response time remains almost constant, and (ii) the average response time significantly increases once the number of clients exceeds a given threshold, qualified as scalability threshold, dependent on the considered server configuration. The scalability thresholds for large server configurations when using the dynamic load balancing algorithms cannot be given due to the limitation of our simulation environment. However, it can be theoretically shown that a scalability threshold exists for any server configuration.

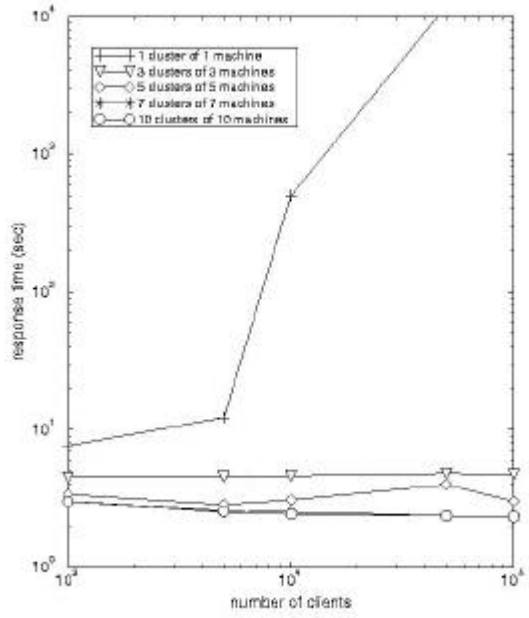

Figure 6: Performance of the PROFILE algorithm

Up to this point, we have given performance results for each of the load-balancing algorithm. Let us now give a comparison among them so as to show the benefits of our profile-based load balancing from the standpoint of the system's scalability (i.e., the number of supported clients at the scalability threshold, for a given configuration) and timeliness (i.e., the server's mean response time for a given configuration). Results of the comparison between DYN and STAT, and between PROFILE and STAT are given Figure 7 and Figure 8.

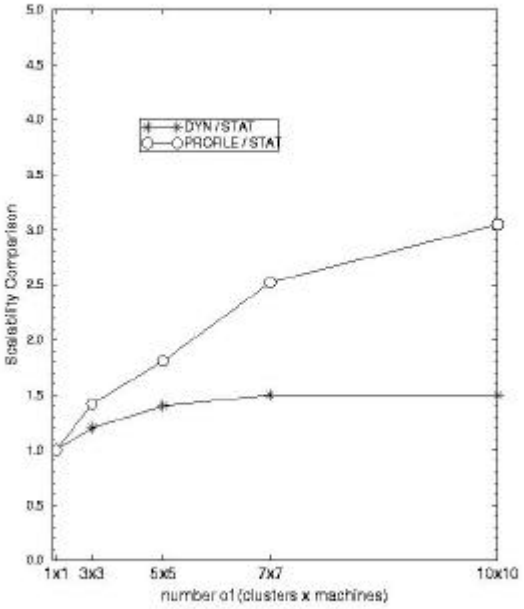

Figure 7: Benefits of the DYN Algorithm -- Scalability

As already raised, our simulation environment did not allow us to perform extensive scalability measures, scalability measures of Figure 7 have thus been evaluated theoretically. As expected, results show that the server's scalability increases as the configuration gets larger. They further demonstrate that our algorithm offers higher scalability than DYN, which itself offers higher scalability than STAT.

Figure 8 gives simulation results regarding comparison of the load balancing algorithms with respect to the server's timeliness guarantees. Results show that DYN and STAT offers similar response times while PROFILE offers far better response times. Furthermore, it can be noticed that the size of the server's configuration has low impact when it exceeds 5 clusters of 5 machines.

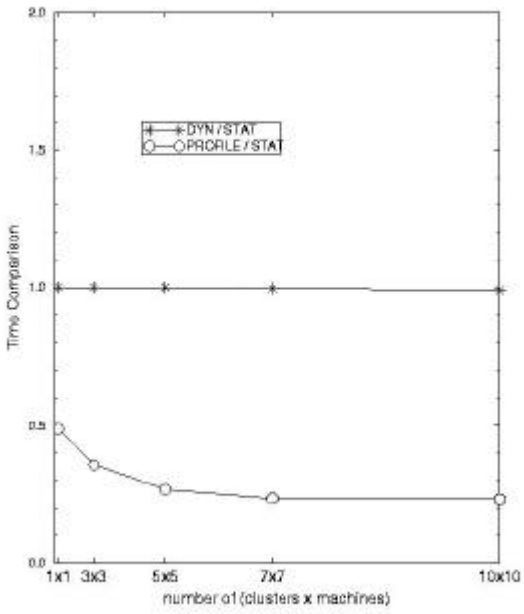

Figure 8: Benefits of the DYN algorithm -- Timeliness

6 Deployment of ETEL++

This section briefly presents the algorithms enforcing the deployment of ETEL++. These algorithms are presented with a finer level of details in the Document DJ4, since the policy they enforce is specific to ETEL++. We therefore recommend the reading of the Section 7 of the Document DJ4.
The following text is a summary of the algorithm that is used by ETEL++ for the deployment.[2]

ETEL++ dynamically checks for the performance of all the servers involved in its distributed architecture, and decides upon the route articles will follow to produce the editions in an efficient way. The performance of servers is obtained by the means of the Service Deployment workpackage. Given the CPU and the network load of each server, each server is asked to perform a particular task, to possibly generate editions of the behalf of heavily loaded servers, or to propagate specific data to other servers.

If a CPU load factor is above a given threshold, then the corresponding machine is said to be CPU bounded. In this case, ETEL++ considers this machine has not enough CPU power to transform articles into the final representations required for user delivery. The same also applies for the network. If the available bandwidth between two machines is scarce, then it is undesirable to transfer between them a large volume of data. In this case, ETEL++ considers the machine at the end of the link to be network bounded. A machine can be solely CPU-bounded, solely network-bounded or both. In the first case, this machine will not transform documents from one media to the other, but will rather try to obtain a copy of the relevant documents after having asked another non-CPU-bounded machine to do the transformations on its behalf. In the second case, a network-bounded machine will rather get a copy of some documents, and subsequently generate all the transformed versions to match users' expectations. ETEL++ assimilates the case of a machine that is both CPU- and network-bounded to the case of machines that are solely CPU-bounded. It is also possible that a machine is not bounded at all. In this case, ETEL++ sends active documents, since this tends to minimize bandwidth consumption. Before initiating the computation of the data flow map, all servers can therefore be classified in NO-BOUNDED, CPU-BOUNDED or NETWORK-BOUNDED. This classification enables the computation of a

[2] This text comes from another document that gives an overview of the technical aspects of ETEL++.

performance sensitive data-flow map in which each server is assigned a specific role, and in which the “journey” of each article is decided.

7 Annex: API of the Monitoring Tools

We present the main classes of the Monitoring Tools. More information can be howerver obtain from http://hyperwav.fast.de/WPG.

7.1 Class Hierarchy

- class java.lang.Object
  - class ServiceDeployment.Monitor
  - class ServiceDeployment.Frequency
  - interface ServiceDeployment.Measurement
  - class ServiceDeployment.MeasurementAggregate (implements ServiceDeployment.Measurement)
  - class ServiceDeployment.basicMeasurement (implements ServiceDeployment.Measurement)
  - interface ServiceDeployment.Value
  - class ServiceDeployment.ValueAggregate (implements ServiceDeployment.Value)
  - class java.util.Observable
    - class ServiceDeployment.View
      - class ServiceDeployment.History
      - class ServiceDeployment.Filter (implements java.util.Observer)
        - class ServiceDeployment.Interpolation
        - class ServiceDeployment.Select
        - class ServiceDeployment.Average
    - class ServiceDeployment.Test (implements java.util.Observer)
  - class ServiceDeployment.Resource
    - class ServiceDeployment.CPU
    - class ServiceDeployment.Disk
    - class ServiceDeployment.Memory

- class ServiceDeployment.Network
- class ServiceDeployment.Target
- class java.lang.Throwable (implements java.io.Serializable)
  - class java.lang.Exception
    - class ServiceDeployment.NoEvaluationException

7.2 Class ServiceDeployment.Monitor

```
java.lang.Object
   |
   +----ServiceDeployment.Monitor
```

public class **Monitor**

extends Object

Monitor Manages the Histories. It requests values of Measurements periodically and delivers them to Histories. Services may also request a single value to Monitor by mean of Monitor.getEvaluation(Measurement m).

See Also:

Measurement, History

`public Monitor ()`

Constructor of Monitor. There should be only one Monitor on a Host.

`public History monitor (Measurement m, Frequency f)`

Method monitor allows an application to get an History and to start the updating of the History with the provided frequency.

`public void stopMonitoring (History h)`

Method stopMonitoring allows an application to stop the updating of the History. This method does not destroy the History.

7.3 Class ServiceDeployment.basicMeasurement

```
java.lang.Object
   |
   +----ServiceDeployment.Measurement
           |
```

```
          +----ServiceDeployment.basicMeasurement
```

public class **basicMeasurement**

extends Measurement

`public basicMeasurement (Resource r, Target t)`

Constructor allows the application to define the Resource and Target to monitor.

`protected Value getValue ()`

Applications do not access this method which is the way Monitor updates an History.

Overrides:

getValue in class Measurement

`public Resource getResource ()`

Delivers the Resource specified in Measurement.

`public Target getTarget ()`

Delivers the Target specified in Measurement.

7.4 Class ServiceDeployment.View

```
java.lang.Object
   |
   +----java.util.Observable
           |
           +----ServiceDeployment.View
```

public abstract class **View**

extends Observable

superclass of History and Filter. View defines the interface applications can access on a Request-based mode. View provides a set of methods allowing to evaluate the value stored in an History in a given time range.

See Also:

History, Filter

`public abstract Value getEvaluation (Date date) throws NoEvaluationException`

Delivers the Value corresponding to a given Date. Type of Date is defined in Java.util.Date.

```
public abstract Date getBegin () throws NoEvaluationException
```

Delivers the Date corresponding to the first Value.

```
public abstract Date getEnd () throws NoEvaluationException
```

Delivers the Date corresponding to the last Value.

```
public abstract boolean isEmpty ()
```

Delivers true if View does not store any Value.

```
public abstract Value getValueAfter (Date date) throws NoEvaluationException
```

Delivers the first Value of View after Date.

```
public abstract Value getValueBefore (Date date) throws NoEvaluationException
```

Delivers the first Value of View before Date.

```
public abstract Date getDateAfter (Date date) throws NoEvaluationException
```

Delivers the first Date of View after Date.

```
public abstract Date getDateBefore (Date date) throws NoEvaluationException
```

Delivers the first Date of View before Date.

```
public abstract String getType ()
```

7.5 Class ServiceDeployment.Test

```
java.lang.Object
   |
   +----java.util.Observable
           |
           +----ServiceDeployment.Test
```

public abstract class **Test**

extends Observable

implements Observer

Test is applied on the output of History or Filter in order to provide notification-based interface.

See Also:

Measurement, History

```
public Test (View v)
```

Constructor of Test. V is the View Test has to be plugged on.

```
public abstract boolean evaluate ()
```

implements the test to apply on View. If evaluate delivers true the service is notified.

7.6 Class ServiceDeployment.History

```
java.lang.Object
   |
   +----java.util.Observable
           |
           +----ServiceDeployment.View
                   |
                   +----ServiceDeployment.History
```

public class **History**

extends View

History gathers values delivered by Measurement. Each value is associated to the corresponding Date. History provides the interface of View in order to retrieve a value before or after a date. Filters and Tests may also be applied on the output of History in order to provide adequate output. There is no public interface to History since History is delivered and updated by Monitor. The only way to interrogate History is specified in the View interface.

See Also:

Filter, Test, Value, Monitor