Automatic Wide-Area Deployment and Caching of Component-Based Services

Peter Backx, Bart Dhoedt, Filip De Turck, and Piet Demeester.

Abstract—Component based frameworks, such as J2EE and .Net, reduce complex software systems into manageable components. Whereas this simplifies the development process, the deployment phase is still a challenge. Currently the only manageable way of deploying an application is to a single server or a local area cluster. This approach forces companies to own and manage their individual server farm and thus introduces a lot of overhead. Furthermore servers are usually at the edge of the network far away from customers, which possibly leads to several problems, amongst others uncontrollable delays in the delivery of the service, because of network congestions, router outages or processing bottlenecks on the server side.

This paper goes into the possibility of automatically deploying components on a wide area network. Several issues are addressed: A framework is presented to identify suitable components for this automatic deployment. Next, a heuristic is discussed to find strategic locations for component placement. While it is possible to simply deploy every component at every server, a finer level of control leads to improved results and better usage of both the networks and the servers' resources.

Index Terms—Web caching, Component-based services, Distributed systems.

I. INTRODUCTION

Web caching has traditionally been used to great effect on static web pages [1]. By employing a proxy cache such as Squid [2], the outgoing traffic can be reduced and the user perceived latency can be shortened. Furthermore, servers are less loaded, because the handling of part of the requests is distributed over the network and the server itself receives fewer requests. Also the uptime of servers can appear higher, because pages are still available in the cache although the server might be down.

However, static proxy caching has hit a roadblock with the introduction of the latest web applications that rely on extremely dynamic content, adapted on a per-user basis. In return, content distributors such as Akamai have “activated” their caches. With their Edge Side Includes (ESI) [3] and the work being done in the IETF Open Pluggable Edge Services (OPES) [4] working group, caches can now actively change the cached pages on an as needed and per-user basis. Examples range from simply inserting localized add-banners to delivering fully personalized and themed pages.

The disadvantage of ESI, however, lies in the fact that it introduces a whole new set of tags to learn and that it tends to obfuscate the distinction between the user’s view and the controller executing the business logic of an application, an important aspect of today’s model-view-controller (MVC) software architectures. The MVC pattern [5] is one of the most used and most successful approaches in order to keep the development of large software projects manageable. Frameworks such as e.g. Struts [6] and WebWork [7] are widely used to support the MVC pattern in conjunction with J2EE [8] application servers.

One way to solve the inherent problems with ESI is to do away with the special tags and to directly use the components of applications as the unit of caching. These components (such as the Enterprise Java Beans in J2EE) are perfectly suited for distributed deployment and Akamai’s EdgeComputing [9] is a first step in this new evolution. However, this still requires that the developer explicitly splits the application into a part that will be deployed on the edge servers and one that stays on the servers of the application provider. The enterprise code is then exposed as a web service to the edge.

The logical next step in this evolution is the fully automatic deployment of cacheable components in a content distribution network. To reduce erroneous deployments that have unfavorable effects on performance by lack of experience or knowledge, cacheable components will also need to be automatically identified. Based on the load on the individual components of a service, decisions need to be made to cache one or more components of an application inside the network.

In order to analyze the different factors and strategies that come into play, we first describe the components of the J2EE architecture in more detail. While we focus on J2EE in this paper, components with similar behavior are present in other frameworks.

The Java 2 Enterprise Edition platform is based on a multitiered architecture as shown in figure 1. Clients connect either through a web browser to the web tier or with an application client directly to the business tier. The J2EE server implements both the web and business tier. The web tier consists of Java Server Pages (JSP) and servlets while the business tier is composed of Enterprise Java Beans. The Enterprise Information System tier handles the enterprise information software, such as databases and enterprise resource planning. As such, an application based on J2EE technology is already “network-aware” because different components can be deployed on different servers. However, if
this deployment is done on an ad hoc basis or possibly with poor judgment on a wide-area network, a lot of network traffic will be generated and large delays will be introduced. Consequently this sort of distributed deployment is currently only feasible on a fast local area network (a server cluster).

Reference [11] proves that it can be beneficial to distribute components of the same application over a wide-area network. However, doing this manually is very time consuming and requires a thorough knowledge of the application. Indeed, when considering distributed deployment on a wide area network, it is important to consider the additional network traffic that could be generated. The different J2EE components behave differently, so a distinction should be made. Table I gives an overview of the different J2EE components and their state properties.

Stateless components are ideal candidates for optimizing network traffic, because they can be moved freely around in the network. However, also components with session state could be cached. Some care is needed, though, because requests for the same session should always be directed towards the correct component. Entity beans, with shared state, are usually linked to a database and should always be up-to-date and are thus harder to cache and distribute, however a number of schemes have been developed that facilitate caching database query results [12]. Especially when using read-mostly entity beans, caching can lead to major improvements in response time and network traffic. Read-mostly beans can, for instance, be found in any online shop: A customer will be browsing and reading a catalogue, while updates to that catalogue are much less frequent in comparison to the number of requests made by users.

This paper is structured as follows: Section II gives a short overview of the framework that was developed for modeling components and identifying candidates for remote deployment. Section III shows how network traffic can be optimized by distributing components over a regular tree network. This topology limitation is removed in section IV with a heuristic for automatic and adaptive deployment of components on a ny network. Finally, section V presents results obtained by simulating the heuristic.

II. COMPONENT MODEL

In order to make informed decisions we need a generic model of a component-based application. We propose the use of a component dependency graph (CDG) to represent the application. This CDG is related to the WebGraph proposed in [10], but is enhanced and adapted for our needs. Figure 2 shows an example of some components of an online shopping application. All information regarding network traffic going in and out of a component on requests is represented as well as the type of states (not shown is their update frequency).

Next to this static information of the application, also dynamic information is needed, namely the rates at which clients request the component. These can be obtained by monitoring the network traffic, which can be done by the caches themselves at runtime.

With this information we can identify components or clusters of components that are useful for deploying at a remote site. A good metric for this is the ratio between traffic towards the client and traffic towards other components or servers (that are not part of this cluster). This is what we call the efficiency of a cluster C:

\[
E(C) = \frac{\sum_{n \in [\text{cluster}]} S_n \lambda_n}{\sum_{n \in C} \text{output}(n, C) \lambda_n + \sum_{n \in C} \text{output}(s, C) \mu_s}
\]

(1)

With:

- \(S_n\), the number of bytes that is exchanged between a component \(n\) and its client (this is indicated on the figure by the number next to the arrows pointing from client to component).
- \(\lambda_n\), the average external request frequency for component \(n\). This only includes requests from outside
the cluster, so it does not contain requests between two components in the cluster.

* \( \mu_s \) the average update frequency of state \( s \).

* \( \text{output}(s,C) \) the traffic (per request or per update) going out of the cluster, either as a result of a request (when \( x \) is a component) or as a result of state updates (when \( x \) is a state).

A high efficiency indicates that there is a lot of traffic between the clients and the cluster and little traffic between the cluster and other components or servers. A cluster with high efficiency is therefore a good choice for deployment closer to the clients.

III. CACHING IN A REGULAR E-ARY TREE

This section investigates component placement on a regular tree-shaped network: One server is connected to exactly \( E \) caches, which in turn are also connected to \( E \) caches and so on. The load is considered to be symmetric, thus all clients generate requests for a certain component at the same rate. This also means that all caches situated at the same level in the caching tree will see the same request frequency for that component. This symmetric assumption influences the decision process: if it is decided to deploy a component at a certain level, that component should also be deployed at all the other caches at that level.

We can divide the network traffic generated by the requests and their answers into three parts: Firstly the traffic between clients and the cache (where the component is cached that is requested), secondly the direct traffic that is generated by the component towards others to formulate the answer and thirdly the traffic for keeping the states up-to-date. We can now group these traffic costs into two groups: the request costs which is directly dependent on the traffic generated by requests, this includes both the client-cache and the direct inter-component traffic and the refresh cost which is associated with keeping the states up-to-date.

Consider now a regular \( E \)-ary tree with clients \( d \) hops away from the server as illustrated in figure 3. The request cost for caches at a certain level \( 1 \) (i.e. \( l \) hops away from the server) can then be easily calculated from the difference between incoming and outgoing traffic as a result of the requests. With the symbols as defined in the previous sections we get

\[
C_n^{req} = \left( S_n^1 (d-1) + \text{output}(n,C) \right) \lambda_n E^d.
\]  

(2)

Where the request rate of individual clients is \( \lambda_n \) and we leave all the components that are not part of the cluster on the server.

For the traffic generated by state updates, we will assume for now that caches get state updates directly from the server. If caches get state updates from the server, the cost for refreshing is proportional to the number of hops between cache and server. The total refresh cost is

\[
C_n^{refresh} = \text{output}(s,C) \mu_s E^l.
\]

(3)

Figure 4 shows the costs for an example with one component and one state in 4-ary tree with 6 levels. Clearly the best caching location is at the point where the refresh and request costs are balanced. This observation is what leads us to the development of a heuristic for deciding the caching location in a random network.

IV. HEURISTIC FOR GENERAL NETWORKS

As explained in the previous section suitable caching locations can be found at points where the request cost (that goes up towards the server) and the refresh cost (smaller closer to the server) are in balance. While any cache can calculate these costs, it cannot autonomously assess the consequences of deploying at a specific location. We propose a heuristic based on pushing components (or clusters thereof) from the server into the network. This way, components will gradually evolve towards their most optimal caching location.

The problem we address here is the following: A cache (or the server) already caching one or more components needs decision criteria for pushing one or more components towards its child caches (in the distribution tree). The difference in network traffic will be solely on the links between the parent cache and its children. If a component \( n \) is cached at the
parent the traffic consists of requests only and their replies (with \( \lambda_n \) the request rate seen by the parent cache), namely

\[
\sum_{n \in C} S_n'\lambda_n.
\]

(4)

In the other case when cached at the children we have both traffic towards dependent components and state update traffic (with \( t \) the distance in hops between parent cache and server):

\[
\sum_{n \in C} output(n, C)\lambda_n + \sum_{s \in C} (1 + (E - 1)(t + 1)) output(s, C)\mu_s
\]

(5)

With this in mind the parent cache can now decide to push the cluster under consideration if

\[
\sum_{n \in C} output(n, C)\lambda_n + \sum_{n \in C} (E - 1)\mu + E output(s, C)\mu_s < \sum_{n \in C} S_n'\lambda_n
\]

(6)

\[
\Rightarrow \frac{\sum_{n \in C} (S_n' - output(n, C))\lambda_n}{\sum_{n \in C} output(s, C)\mu_s} > (E - 1)\mu + E
\]

This defines an easy to evaluate threshold and a distributed heuristic that can be run by all participating caches. Caches can either use this for the components they know of (by monitoring traffic) or they can obtain the full CDG from the server to optimize placement for the whole application.

V. SIMULATION

The example shown in figure 2 was used in a simulation of the heuristic on a 500 node network. The network was generated using the generalized linear preference model [13] and resembles the topology of the Internet (although obviously at a much smaller scale). The results presented focus on the Product component and its caching location. The popularity of the Cart component and the size of the Product Database\(^1\) were varied. The popularity is expressed as a percentage, indicating the fraction of requests destined for the Cart component. Both the caching location (in average number of hops between server and Product component caches) and the network cost (in number of bytes transferred over the network) are monitored and plotted.

Figure 5 shows the caching location of the Product component. If its state is small, it can be cached closer to the clients because the traffic generated by state update will also be small. If it is popular it can also be cached closer to the clients because the reduced request traffic will outweigh the

\(^1\)In a real application this would only be a partial view of the actual Product Database.

increased refresh traffic.

Figure 6 presents the network traffic when no caching is used and with caching as a function of the size of the Product Database state. This size does not affect the network traffic when no caching is used, however minor variations are due to the randomness of the request streams. With caching a major reduction in network traffic can be obtained. For a smaller state the network traffic is slightly lower because the state can be cached closer to the clients, however even for large states the benefit of caching is clear.

VI. FUTURE WORK

A number of enhancements can be made to the proposed framework and heuristic and are currently under investigation, most importantly:
Currently, state updates can waste a lot of bandwidth because every cache updates its state with the server, while it might be beneficial to update the state with another cache. Therefore more optimal techniques are under research. The most obvious way would be to set up an IP multicast tree between the different nodes that cache a state. However IP multicast is currently not widely deployed.

The current framework and heuristic have only been evaluated for components that reside on a single server. Thorough testing of their behavior in a multi-server environment will be undertaken.

VII. CONCLUSION

This paper presented a framework for analyzing component based applications and for generating a component dependency graph (CDG). With this CDG clusters can be identified that could benefit from caching inside the network (for instance on proxy caches). A distributed and adaptive heuristic was then proposed to move components automatically towards their most suitable caching locations. A number of simulation results were presented to support this claim.

REFERENCES

This set of volumes contains papers presented at the International Conference on Internet Computing (IC'04) and International Symposium on Web Services & Applications (ISWS'04). Their inclusion in this publication does not necessarily constitute endorsements by the editors or by the publisher.

Copyright © 2004 by CSREA Press

Copyright and reprint permission:

Copying without a fee is permitted provided that the copies are not made or distributed for direct commercial advantage, and credit to the source is given. Abstracting is permitted with credit to the source. Contact the editors or the publisher, for other copying, reprint, or republication permission.


Printed in the U.S.A.