Available online at www.sciencedirect.com

SCIENCE DIRECT®

Computer Communications 28 (2005) 1028–1037

computer communications

www.elsevier.com/locate/comcom

An efficient load balancing strategy for scalable WAP gateways

Te-Hsin Lin, Kuochen Wang*, Ae-Yun Liu

Department of Computer and Information Science, National Chiao Tung University, 1001 Ta Hsueh Rd., Hsinchu 300, Taiwan, ROC

Received 13 May 2003; revised 25 November 2004; accepted 7 December 2004
Available online 20 December 2004

Abstract

Clustering provides a promising way to build a scalable, reliable, and high-performance WAP gateway architecture. However, this requires an efficient load balancing mechanism for assigning a request to a suitable gateway in the cluster, that can offer the best service. In addition, unpredictable connection time and nonuniformity of incoming load from different mobile clients are big obstacles to load balancing among real gateways. In this paper, we propose a load balancing strategy that has the following features: (1) estimating the potential load of real gateways with small computation time and no communication overhead, (2) asynchronous alarm sent when the utilization of a real gateway exceeds a critical threshold, and (3) WAP-awareness. We also propose a scalable WAP gateway (SWG) architecture that consists of a *WAP dispatcher* and a *cluster of real gateways*. The WAP dispatcher is a front-end distributor with our load balancing strategy. To prevent the WAP dispatcher from becoming a bottleneck, the WAP dispatcher distributes mobile clients' requests in kernel space and does not process outgoing gateway-to-client responses. Experiment results show that our SWG has better load balancing performance, throughput, and delay compared to the LVS and the Kannel gateway.

Keywords: WAP gateway; Load balancing; Clustered architecture; Asynchronous alarm

1. Introduction

The WAP gateway has been introduced by the WAP Forum [1] to provide efficient wireless access to the Internet. It performs protocol translation and content conversion to reduce the size of data transmitted over the wireless network. The growth of the demand for wireless data services will raise great challenges in terms of performance, scalability and availability of the WAP gateway. Since protocol translation and content conversion are computation intensive, a monolithic gateway will become overloaded and is not sufficient to conquer these challenges. An efficient way to cope with the growing demand is adding extra hardware resources to provide clustered architecture. Upgrading the gateway with a faster model usually results in high price/performance and can not scale up with the demand. That is, turning a monolithic WAP gateway into a clustered gateway is a cost effective alternative to build a scalable, reliable, and high-performance WAP gateway. However, it requires an efficient load balancing mechanism for assigning a request to a selected gateway in the cluster, that can offer the best service. In the WAP model, a WAP client needs to establish a connection with a WAP gateway before it can access Internet resources. The WAP gateway maintains the connection state information until the connection is terminated. In the connection period, the client can make unlimited number of requests to the WAP gateway. Both unpredictable connection time and nonuniformity of incoming load from different mobile clients bring challenges to load balancing among gateways in the cluster. These peculiarities may result in some gateways in the cluster having heavy load, but some with light load.

The WAP standard [1] has suggested that the assignment decision can be taken at the client side through random selection of several configured gateways, or at the gateway side through a WSP redirection mechanism [4] to redirect a connection request from an overloaded gateway to a light-loaded one in the cluster. These two mechanisms have some drawbacks. The client side approach is not scalable and may result in a skewed load on a gateway if too many clients use the same gateway. The WSP redirection mechanism,

* Corresponding author. Tel.: +886 3 5131363; fax: +886 3 5721490.
E-mail address: kwang@cis.nctu.edu.tw (K. Wang).

0140-3664/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.comcom.2004.12.001

achieved by returning the address of the new selected gateway instead of returning connection success, provides scalability and availability. However, it increases the wireless network traffic and response time because of the continued message exchange [4].

A clustered architecture is composed of real gateways. It usually integrates a front-end distributor, which is a load balancer and is used to forward a mobile client's request to a proper real gateway. The front-end distributor is transparent and provides a single user-view interface to a WAP client. The Kannel gateway [3] is based on this architecture. This architecture can provide scalability and load balancing among real gateways. However, the main problem of this architecture is that the front-end distributor may become a bottleneck under heavy request load.

In this paper, we design and implement a *scalable WAP gateway* (SWG) which is also a clustered architecture. The SWG consists of a group of real gateways connected by a fast interconnection network. It integrates a kernel-level front-end distributor, called *WAP dispatcher*, which is different from the front-end distributor of the Kannel gateway. The distributor of the Kannel gateway is running at the application level and it modifies incoming requests and outgoing responses. To prevent the WAP dispatcher from becoming a bottleneck, our WAP dispatcher forwards incoming request to a proper real gateway in kernel space and does not handle gateway-to-client responses. This is particularly useful for WAP services that are characterized by small sizes of client requests that may generate large sizes of responses. The WAP dispatcher uses the Weighted Round Robin (WRR) scheduling algorithm [6] which uses weights provided by the load balancing algorithm to allocate connections among real gateways in the cluster. Our load balancing algorithm has the following features:

- Estimating potential load with small computation time and no communication overhead.
- Asynchronous alarm, which is a feedback alarm mechanism, sent when the utilization of a real gateway exceeds a critical threshold.
- WAP-awareness, which is useful in performing load balancing based on WAP request types.

The paper is organized as follows. We review existing approaches in Section 2. The design of the proposed load balancing mechanism is presented in Section 3. In Section 4, the performance of the proposed mechanism is evaluated. Finally, Section 5 gives concluding remarks and future work.

2. Existing methods

2.1. Linux virtual server

The Linux Virtual Server (LVS) [6] is a highly scalable server built on a cluster of real servers, with the load balancer running on the Linux operating system [7]. The architecture of the cluster is transparent to end users. The end users only see a single virtual server. A real server can be any server, such as a WAP gateway or an HTTP server. The real servers may be interconnected by a high-speed LAN or by a geographically dispersed WAN. The front-end of the real servers is a load balancer, called *LinuxDirector*, which distributes requests to different servers and makes parallel services of the cluster to appear as a single virtual service using a single IP address [6]. Scalability is achieved by transparently adding or removing a real server in the cluster. The LVS uses the Weighted Round-Robin (WRR) and Weighted Least-Connection (WLC) scheduling algorithms to allocate session connections among real servers.

2.2. Kannel gateway

The Kannel gateway is an open source WAP gateway with built-in load balancing capability. It is based on a clustered architecture and consists of two components: a *bearer box* and a group of *WAP boxes*. The bearer box, a front-end distributor, provides a unified interface to the WAP clients and implements the WDP layer of the WAP protocol stack. The WAP box runs the upper layers (e.g. WTP and WSP) of the WAP protocol stack. The bearer box will rewrite the WAP request and passes it to the selected box among the WAP boxes via TCP. The bearer box uses the *random* scheduling algorithm to allocate session connections among the WAP boxes. The drawback of the Kannel gateway is that rewriting is done at the application level, and both request packets and response packets need to be rewritten by the bearer box.

3. Design approach

3.1. Overview of the proposed scalable WAP gateway

We propose a scalable WAP gateway (SWG) that can provide system scalability coupled with load balancing capability in the presence of the explosive growth of WAP service. The SWG architecture is shown in Fig. 1. It consists of a *WAP dispatcher* and *N* homogeneous *real gateways* that provide the same WAP services. The SWG uses one public IP address to provide a single interface for WAP clients. Each real gateway can employ any WAP gateway software available on the market. This implies that our architecture is transparent to not only WAP clients but also real gateways. This is different from the Kannel gateway which is transparent to WAP clients but requires to configure specialized software on each real gateway.

For a WAP packet that represents a new WAP connection request, the WAP dispatcher chooses a real gateway from the cluster, and then forwards the packet to that gateway. Subsequent client-to-gateway WAP packets for an existing connection are forwarded to the same gateway.

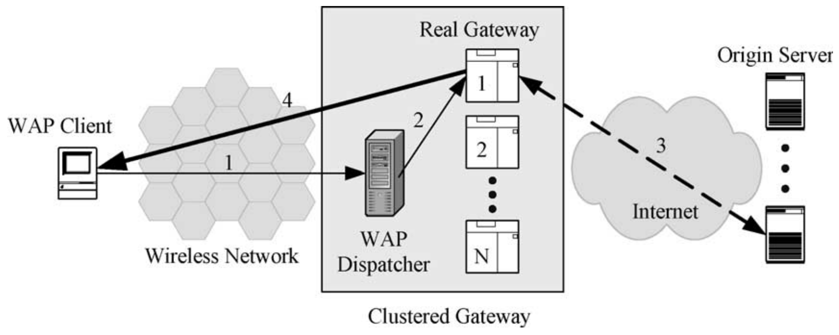

Fig. 1. Scalable WAP gateway architecture.

Outgoing gateway-to-client packets do not need to flow through the WAP dispatcher but are directly sent to the WAP client. This half-connection forwarding method [5] is particularly useful for WAP services that are characterized by small sizes of client requests that may result in large sizes of responses. That is, the WAP client packets are typically very short, e.g. requests to get a new content, and acknowledgements for the data received. Gateway-to-client packets are usually large, as they include encoded content data. The WAP dispatcher handles only incoming packets, and real gateways send the responses directly to the clients and are not aware of the existence of the WAP dispatcher. In contrast, the front-end distributor of Kannel gateway must handle and modify both incoming and outgoing packets in user space, which makes the distributor a bottleneck.

The WAP dispatcher, modified from the LinuxDirector in LVS [6], is a WAP-aware software router. The characteristic of WAP-aware is useful in achieving better load balance performance. Fig. 2 shows the architecture of our WAP dispatcher. The *network interface* includes the network hardware and the device driver, which can transmit and receive packets over a network. The *IP input* module deals with the received IP packets, either delivering them to the upper *TCP/UDP* module or to the *Executor* module if they are WAP packets. The Executor module manipulates the *connection table* and the *destination table*, which will be described later, to forward WAP packets. The *IP forward* module and *IP output* module play the same role to ask the network interface to transmit the outgoing IP packet, but the IP output module needs to create a new IP packet for outgoing data first. The TCP/UDP module provides two transport services: TCP and UDP. The *socket interface* is an interface between the application layer and the transport services, which can let an application interact with the transport services. The *Manager* module manages the destination table and implements the proposed load balancing mechanism. The *timer* module checks whether any entry in the connection table is expired.

When the WAP dispatcher receives an IP packet from the network interface, the kernel calls the IP input module. Modifications are made to this module to be able to handle WAP packets. In this module, the IP packet is examined. If it contains a UDP datagram and the UDP destination port is a port listed in Table 1, we know that the IP packet is a WAP packet and the kernel calls the Executor module to handle it. If the IP packet is not a WAP packet, the kernel calls the TCP/UDP module and the socket interface to handle this packet. After the Executor decides the destination of the WAP packet, the kernel calls the IP forward module to forward the WAP packet to the selected real gateway.

The operation of the WAP dispatcher is based on the following two data structures that hold fundamental information for forwarding a WAP packet: *destination table* and *connection table*. They are illustrated in Fig. 3. Each entry in the destination table represents one real gateway in the cluster. The fields of each destination entry in the destination table

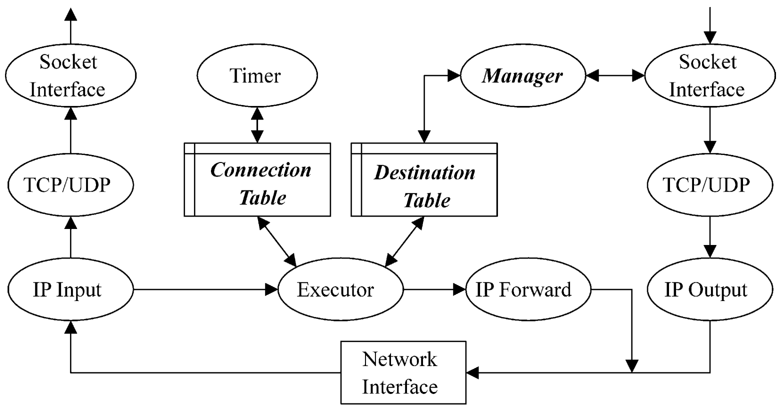

Fig. 2. Architecture of the WAP dispatcher.

Table 1
WAP services

Port number	Service/Protocol
9200	WAP connectionless session service Protocol: WSP/**Datagram**
9201	WAP session service Protocol: **WSP**/WTP/Datagram
9202	WAP secure connectionless session service Protocol: WSP/**WTLS**/Datagram
9203	WAP secure session service Protocol: WSP/WTP/**WTLS**/Datagram

include the *IP address* and the *port number* of the real gateway, a *current weight*, a *connection count*, and a *request count*. Each destination entry is identified by the IP address and the port number of a real gateway. The current weight is used by the WRR to allocate new connections. The request count is the total number of the requests handled by the real gateway. The connection count is the number of active connections present in the real gateway. Any real gateway whose current weight equals to zero will not be allocated any new connections. The WAP dispatcher maintains and manipulates the connection table for forwarding a WAP packet to the selected gateway. Each entry in this table is hashed by the *client IP address* and the *client port number*, and also includes the following fields: the *real gateway IP address* and the *real gateway port number*, and a *timeout* to indicate when the connection entry will expire.

3.2. The Executor

The Executor is a kernel-level extension to the TCP/IP stack. Fig. 4 shows the Executor procedure. The Executor, which realizes the feature of WAP-awareness, inspects an incoming WAP packet and deals with it either discarding or forwarding it to a real gateway for a new or an existing connection. If the target WAP service is not defined in the WAP dispatcher, or if there is no real gateway currently configured for this service, the packet is discarded. To explain the operations of the Executor, we illustrate the sequence of associated events when a WAP client accesses the Internet through the WAP gateway, and then show how the Executor deals with the packet exchange.

3.2.1. Connection establishment

In Table 1, the protocol in **boldface** letters of each service is the *dominating connection protocol* of the service. We define a dominating connection protocol of a WAP service as a protocol that is responsible for connection establishment and connection termination of the service. When a WAP client wants to access the Internet, the client first chooses one of the WAP services and opens a new connection to the WAP gateway according to the dominating connection protocol of the selected service. If the dominating connection protocol of the selected service is the Datagram protocol, the WAP client sends a packet that contains the *S-Unit-MethodInvoke.req* information to represent that the client wants to establish a new connection to the WAP gateway. If the dominating connection protocol of the selected service is the WTLS (WSP) protocol, the connection request is a packet containing the *SEC-Create.req* (*S-Connect.req*) information (see position (1) in Figs. 5–7).

For each incoming packet, the connection table is consulted to find the corresponding connection information. If no information about this packet is found, the Executor checks whether it represents a new connection request according to the dominating connection protocol. If a new connection request arrives, the Executor creates an entry in the connection table and uses the WRR scheduling algorithm to select a target real gateway for this new connection request.

3.2.2. Subsequent requests

For each incoming packet, a hash value is computed based on the source (client) address and the source port number to find an existing connection information about the packet in the connection table. If no entry is found and the packet does not represent a connection request, the packet is discarded. If the packet is for an existing connection, the Executor forwards it to the selected gateway by invoking the IP forward module. The responses, which produces by the real gateway for the requests, does not flow through the WAP dispatcher but are directly sent to the client (see position (2) in Figs. 5–7).

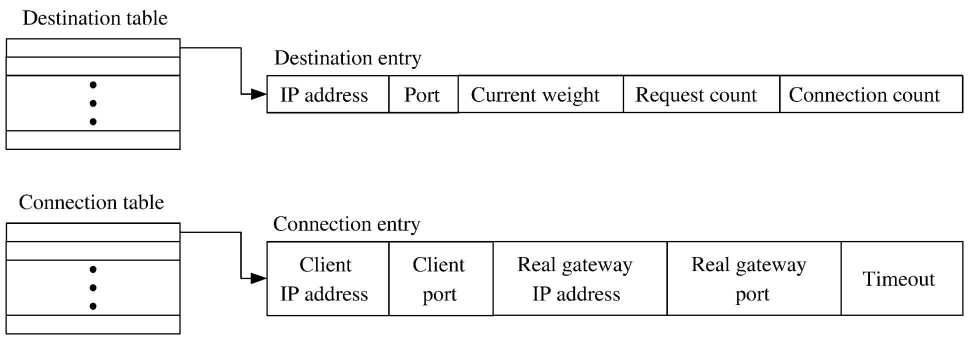

Fig. 3. Data structures of the WAP dispatcher.

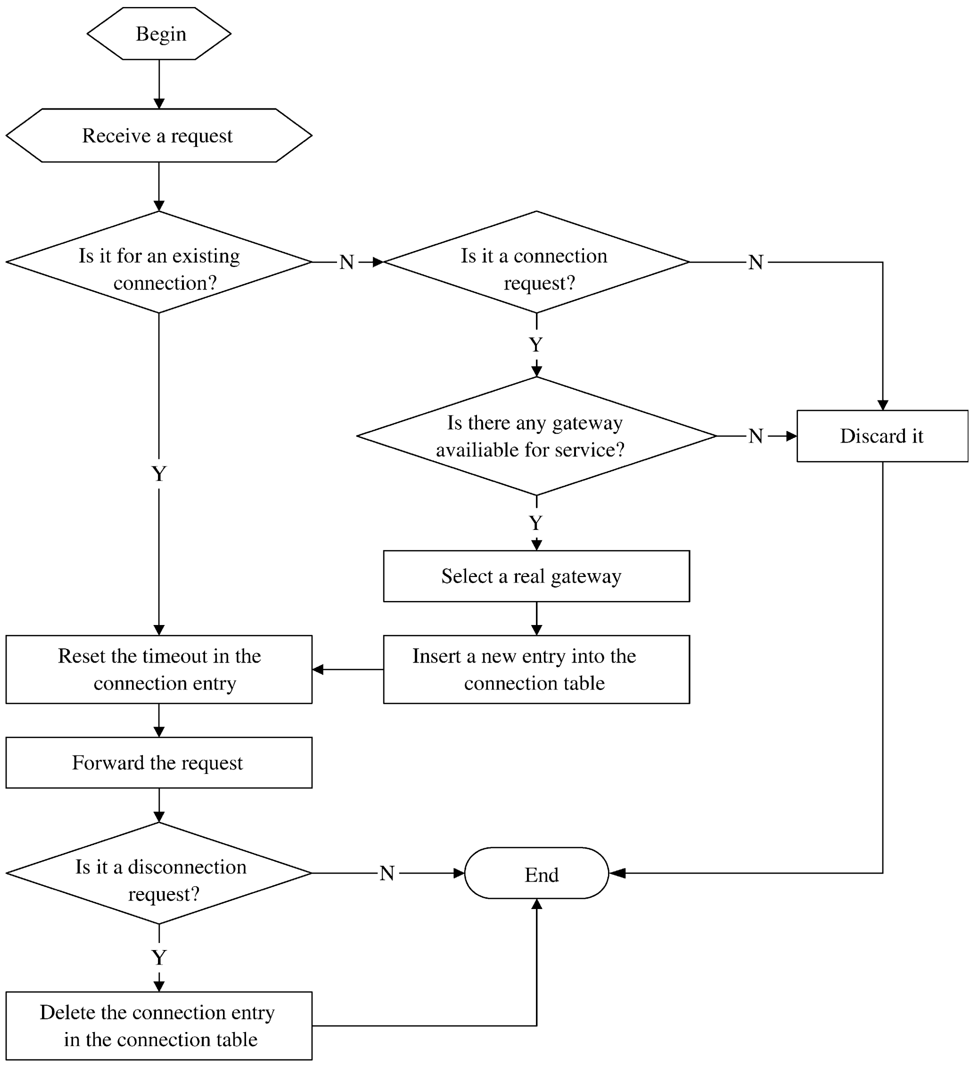
Begin
Receive a request
Is it for an existing connection?
N
Is it a connection request?
N
Y
Is there any gateway availiable for service?
N
Discard it
Y
Y
Select a real gateway
Reset the timeout in the connection entry
Insert a new entry into the connection table
Forward the request
Is it a disconnection request?
N
End
Y
Delete the connection entry in the connection table

Fig. 4. Flowchart of the Executor procedure.

3.2.3. Connection termination

If the Executor receives a disconnection request, according to the dominating connection protocol, the corresponding entry in the connection table will be deleted and the Executor will not forward subsequent requests unless it is a new connection request. If the dominating connection protocol of the selected service is the Datagram protocol, the WAP client sends a packet which contains the *S-Unit-MethodInvoke.req* information to disconnect an existing connection. If the dominating connection protocol of the selected service is the WTLS (WSP) protocol, the disconnection request is a packet containing *SEC-Terminate.req* (*S-Disconnect.req*) information (see position (3) in Figs. 5–7).

If a client loses the connection with its real gateway, the disconnection request may not arrive. This will result in a stale entry in the connection table. To solve this problem, each connection entry has a configurable timeout period. When a request flows through the Executor, the Executor assigns the maximum idle time period to the *timeout* field of the request's corresponding connection entry. The timer module periodically checks and will delete expired entries from the connection table.

3.3. The Manager

The *Manager*, a user-space management module, realizes the load balancing algorithm for allocation of WAP service connections among real gateways based on their load. Table 2 shows the definition of various system parameters used in the load balancing algorithm. The Manager uses two mechanisms for load balancing: (1) estimating potential

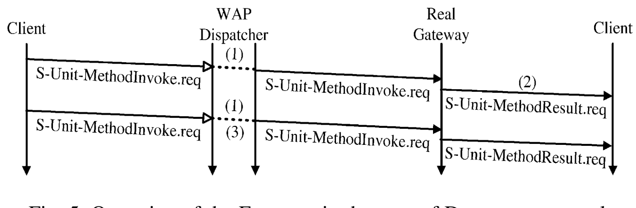

Fig. 5. Operation of the Executor in the case of Datagram protocol.

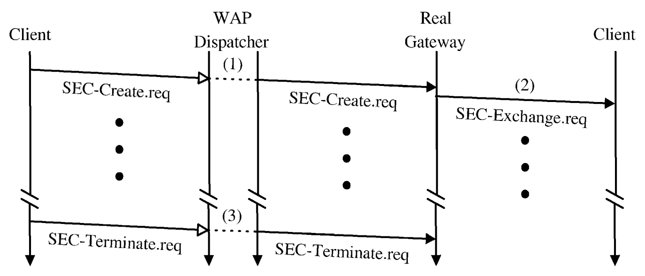

Fig. 6. Operation of the Executor in the case of WTLS protocol.

number of requests, (2) asynchronous alarm, which are described as follows.

3.3.1. Estimating potential number of requests

The Manager performs data collection for each real gateway every T seconds. At the end of the jth measurement period, the Manager computes the number of requests each real gateway i handled, r_i^j, and then estimates the potential number of requests to each real gateway i at next T seconds by [8,10]

$$p_i^{j+1} = \left(\frac{1}{\sum_{m=1}^{K} \alpha^{-m/2}}\right)\left(\sum_{n=1}^{K} \alpha^{-n/2} r_i^{j-n+1}\right),$$

where α is a constant factor and typically is an exponential constant e. We use exponential normalization [8,10] to estimate the potential number of requests, and the observed value of each measurement is combined with a different weight (higher weight for more recent observation) obtained by a decay distribution. The current weight of each real gateway i at next T seconds is computed as follows,

$$c_i = \begin{cases} w_i & \text{if } p_i^{j+1} = 0 \\ \left(\sum_{m=1}^{N} p_m^{j+1}\right) \dfrac{w_i}{p_i^{j+1}} & \text{otherwise.} \end{cases}$$

c_i is assigned to the current weight field of the destination entry for real gateway i in the destination table and used by the Executor to select a proper real gateway for new connection requests.

3.3.2. Asynchronous alarm

Because estimating the potential number of requests does not require tracking or monitoring the actual load condition on the real gateways, a real gateway may become overloaded even if its estimated load is not high. We use the concept of *asynchronous alarm* [11] to provide additional information about the actual load. Each real gateway is allowed to send asynchronous alarms, which signal the beginning and end of an overloading state, to the WAP dispatcher. Thus the WAP dispatcher can make decisions with this feedback information.

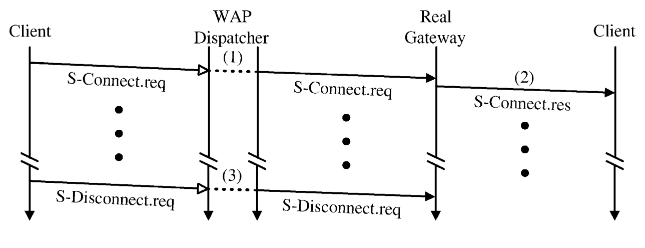

Fig. 7. Operation of the Executor in the case of WSP protocol.

Table 2
Definition of system parameters

Notation	Definition
N	Total number of real gateways
K	Number of last measurements retained
T	Length of the measurement period, in *seconds*
c_i	Current weight of real gateway i
p_i^j	Potential number of requests for real gateway i during the jth measurement period
r_i^j	Number of requests for real gateway i during the jth measurement period
w_i	Default weight of real gateway i

We deploy two user processes on each real gateway. One is a *Web server process* which returns a constant WML content. The other is an *advisor process* [9] which is a fake WAP client periodically sending requests to the real gateway. To get local load information, the advisor process connects to the real gateway, sends a request to retrieve the constant WML content provided by the Web process, disconnects the connection, and then measures the delay. If the delay exceeds a threshold, the advisor process sends an alarm indicating the beginning of an overloading state to the Manager. When the Manager receives the alarm from the real gateway, it then temporarily quiesces the real gateway by assigning the current weight of the real gateway zero. The overloaded gateway is taken off the pool of active real gateways and no new connection is assigned to it. Later on if the delay falls within the given threshold, the advisor process will send an alarm to inform the Manager the end of an overloading state. The Manager then puts the real gateway back to the pool of active real gateways.

4. Evaluation and discussion

4.1. Experiment setup

We compare our scalable WAP gateway (SWG) with the Kannel gateway [3] and the LVS [6] by observing how they perform under heavy request load on a high-speed network. To carry out this, we construct a hardware and software testbed consisting of a *Benclient*, a *Benserver*, and a clustered *WAP gateway*, as shown in Fig. 8. We developed the *Benclient* and *Benserver* to simulate clients and origin servers, respectively.

The Benclient, running on a Linux host, can simulate thousands of concurrent WAP clients. The Benclient is responsible for generating the client-side WAP traffic.

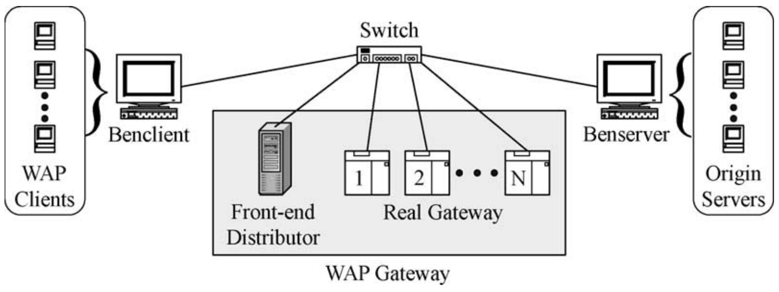

Fig. 8. Experiment environment.

Each simulated WAP client transmits a series of WAP requests to the WAP gateway to download WAP contents on the origin servers. The Benserver, also running on a Linux host, can simulate multiple concurrent origin servers. Each simulated origin server generates HTTP responses according to the WAP gateway requests. Each WAP gateway is composed of a *front-end distributor* and a group of *real gateways*, as shown in Table 3. For the Kannel gateway, the front-end distributor is the bearer box and the real gateway is the WAP box. The front-end distributors of our SWG and the LVS are the WAP dispatcher and the LinuxDirector [6], respectively. The real gateway of our SWG and that of the LVS are modified from the Kannel gateway, named *Standalone Kannel gateway*. The Standalone Kannel gateway removes the built-in load balancing capability from the Kannel gateway and moves the WDP layer from the bearer box to the WAP box. The Standalone Kannel gateway is thus a monolithic gateway handling a client's request independently, like the Ericsson (Nokia) WAP gateway [2].

The experiment environment includes the following machines connecting through 100 Mbps Fast Ethernet Switch: one Pentium MMX 233 MHz serving as a front-end distributor, six Pentium III 500 MHz machines serving as real gateways, one Pentium III 500 MHz machine serving as a Benclient, and one Duron 700 MHz machine serving as a Benserver. The memory size of these machines is 128 Mbytes and the operating system of these machines is the Linux.

4.2. Simulation model and parameters

The parameters of the simulation environment are summarized in Table 4 [13,18] The WAP client in Fig. 1 is modeled as follows. New WAP client session arrivals follow the Poisson distribution [14,18]. A client session arrival corresponds to the time when a client decides to use the WAP service for retrieving WAP contents [12]. The number of consecutive WAP pages which a WAP client requests during a session follows the inverse Gaussian distribution [13,14]. The client's user think time (silent time) between the retrieval of two successive WAP pages is modelled as the Pareto distribution [14,15]. The number of embedded objects per WAP page, including WMLScript and WBMP files, is also modelled as Pareto distribution [14]. In addition, the Origin Server in Fig. 1 is modelled as follows. The number of the simulated origin servers is 200. The delay of the origin servers follows the Exponential distribution [14,16]. The Manager of the WAP dispatcher (in Fig. 1) performs data collection for each real gateway and estimates potential load every 2 s. There are 10 measurements retained for estimating the potential load of each real gateway.

Note that the use of different combinations of distribution will not affect our findings or simulation results in Section 4.3. The reasons are as follows. There are two distinct characteristics of our approach that contribute to the superiority of our approach over the Kannel and LVS. One characteristic is that the backend servers directly send the outgoing gateway-to-client packets to the clients. This can reduce the processing time of the WAP dispatcher. The other characteristic is that the estimation of the potential load of each real gateway can help distribute a request to a proper backend

Table 3
Configuration of each WAP gateway

Gateway	Front-end distributor	Real gateway
Kannel	Bearer box	WAP box
LVS	LinuxDirector	Standalone Kannel
SWG	WAP Dispatcher	Standalone Kannel

Table 4
Parameters of the simulation environment

Category	Parameter	Value
Request generation	Pages per session	Inverse Gaussian ($\mu=7.52$, $\lambda=9.46$)
	User think time	Pareto ($\alpha=1.4$, $k=1$)
	Embedded objects per page	Pareto ($\alpha=1.245$, $k=1$)
Origin Server	Number of servers	200
	Delay	Exponential ($\beta=1$ s)
WAP Gateway	Number of real gateways	$N=6$
WAP	Measurement period	2 s
Dispatcher	Number of measurements retained	10

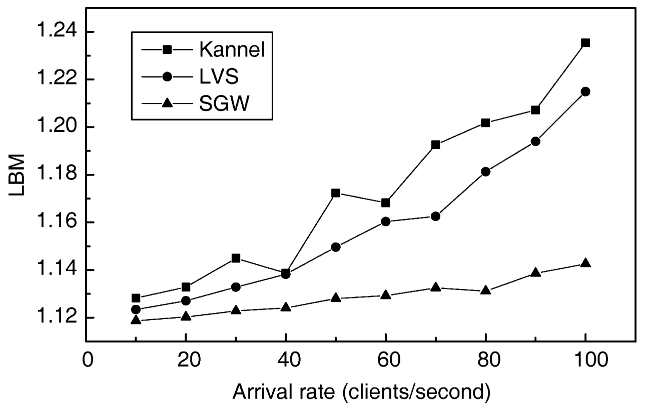

Fig. 9. The effect of client arrival rate on LBM using WAP connectionless session service.

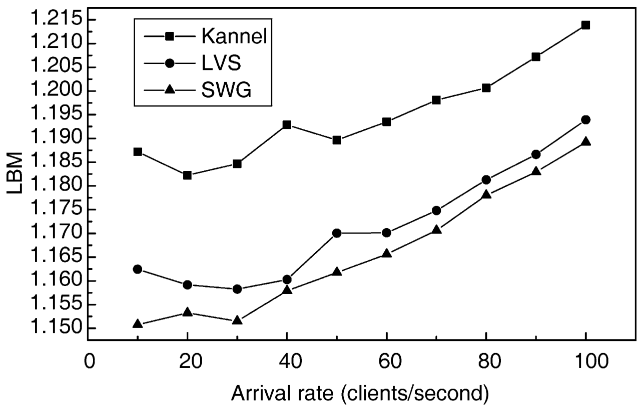

Fig. 12. The effect of client arrival rate on LBM using WAP secure session service.

server with the least load. Such estimation needs very low computation overhead.

4.3. Experimental results

To analyze performance of our load balancing mechanism, we use a metric, *Load Balance Metric* (LBM) [17]. The LBM is the weighted average value of the peak-to-mean ratio that is defined as $peak\_load_j/\sum_{i=1}^{n} load_{i,j}$ where $peak\_load_j$ is the peak load at the jth sampling period and $load_{i,j}$ is the total number of requests received by real gateway i between the $(j-1)$th and jth sampling periods. The definition of the

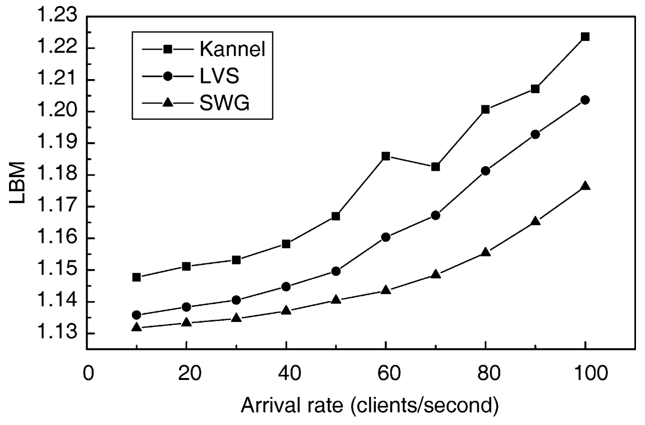

Fig. 10. The effect of client arrival rate on LBM using WAP session service.

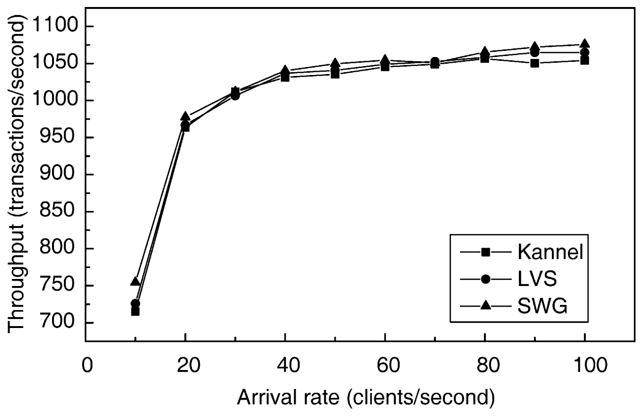

Fig. 13. The effect of client arrival rate on throughput using WAP connectionless session service.

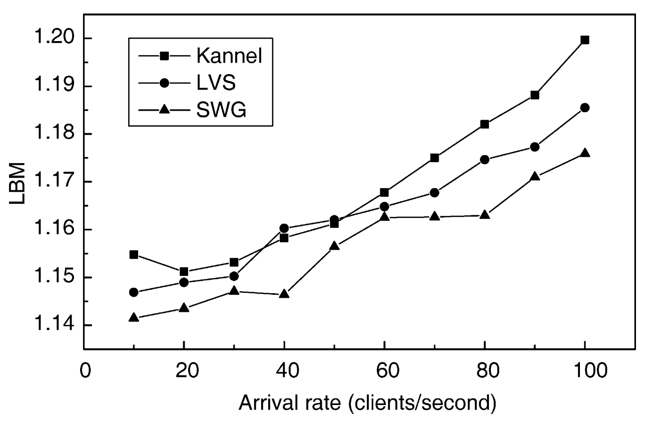

Fig. 11. The effect of client arrival rate on LBM using WAP secure connectionless session service.

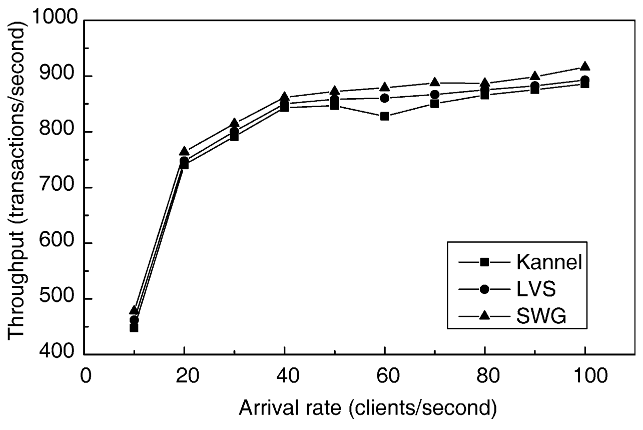

Fig. 14. The effect of client arrival rate on throughput using WAP session service.

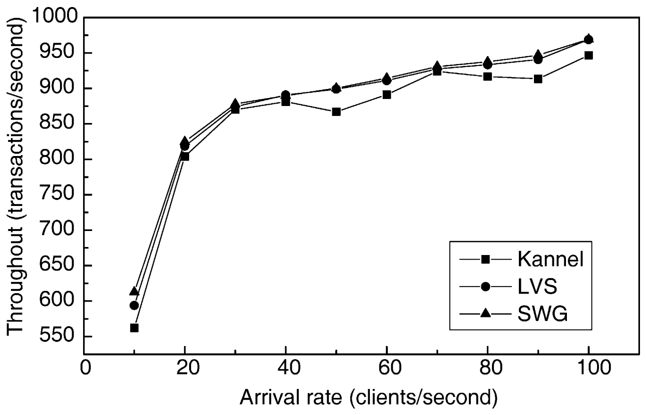

Fig. 15. The effect of client arrival rate on throughput using WAP secure connectionless session service.

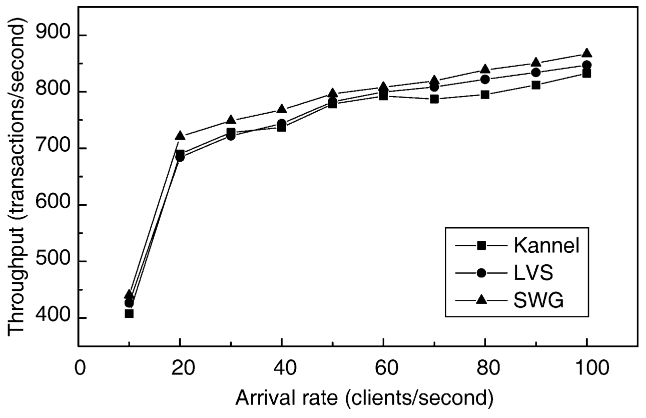

Fig. 16. The effect of client arrival rate on throughput using WAP secure session service.

LBM for a cluster of N real gateways and M sampling periods is:

$$\mathrm{LBM} = \frac{\sum_{j=1}^{M}\left(\frac{peak\_load_j}{\left(\sum_{i=1}^{N} load_{i,j}\right)/N} * \frac{\sum_{i=1}^{N} load_{i,j}}{N}\right)}{\sum_{j=1}^{M}\sum_{i=1}^{N} load_{i,j}/N}.$$

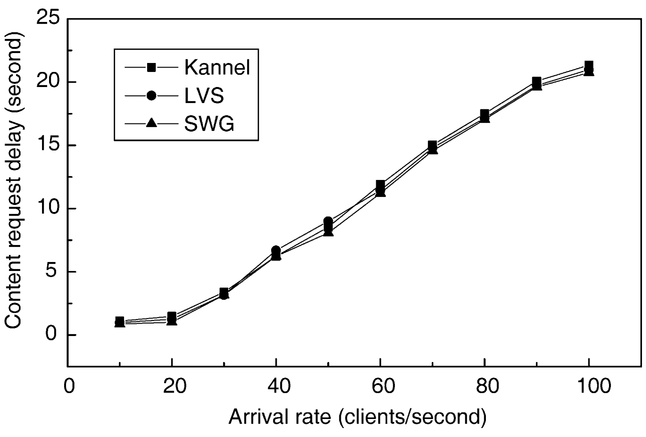

Fig. 17. The effect of client arrival rate on content request delay using WAP connectionless session service.

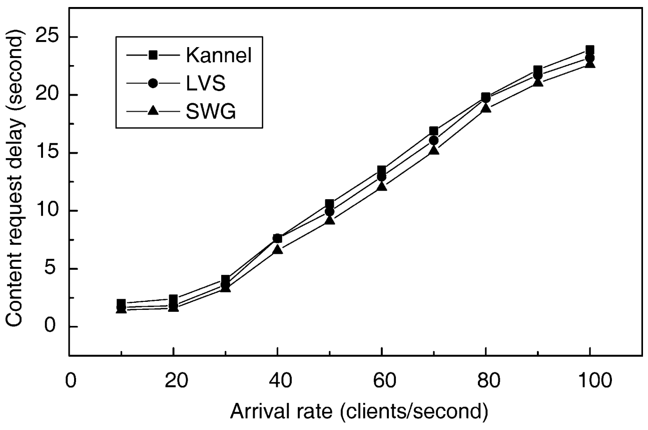

Fig. 18. The effect of client arrival rate on content request delay using WAP session service.

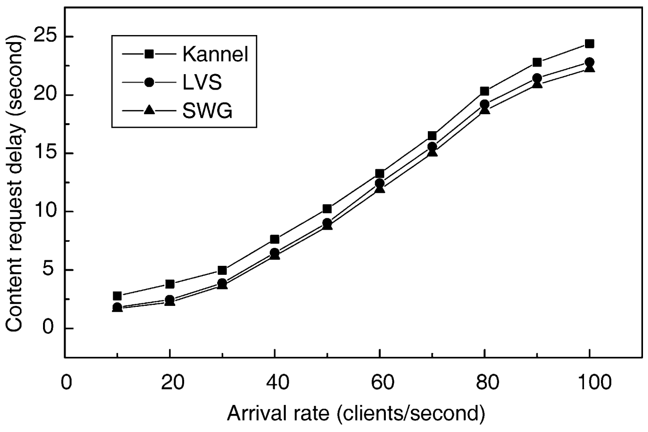

Fig. 19. The effect of client arrival rate on content request delay using WAP secure connectionless session service.

The value of the LBM may range from 1 to at most N, the number of real gateways. Small values of the LBM indicate better load balancing performance (smaller peak-to-mean load ratios) than large values [17].

As shown in Figs. 9–12, our load balancing mechanism (SWG) outperforms the other two methods in terms of LBM,

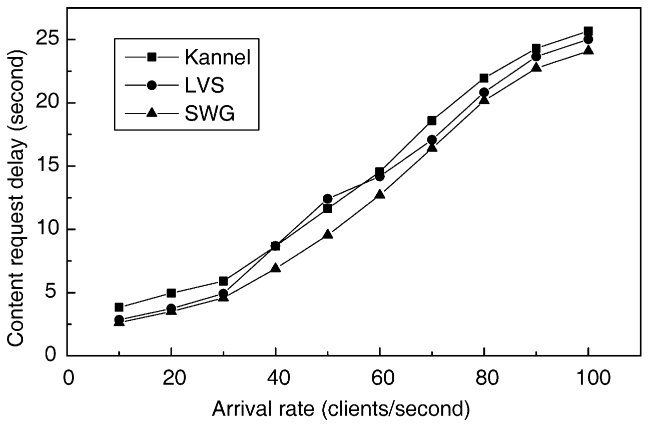

Fig. 20. The effect of client arrival rate on content request delay using WAP secure session service.

especially in the WAP connectionless session service. This is due to that each request from the same WAP clients can be forwarded by the WAP dispatcher to different real gateways. In the Kannel gateway and the LVS, once the front-end distributor decides the served real gateway for a WAP client, the requests from the WAP client will be handled by the same real gateway until the WAP client closes the session. It is hard for them to reassign the requests from the same WAP client to a different real gateway that has light load. As a result, the SWG can provide better load balancing performance compared to the LVS and the Kannel gateways. In addition, the experimental results in Figs. 9–12 have also shown that the use of exponential normalization can estimate potential requests from clients with high accuracy. These results justify the use of exponential normalization. In addition, the exponential normalization is easy to implement and needs low computation time, which can prevent the WAP dispatcher from becoming a bottleneck.

Figs. 13–16 show that the throughput achieved with our SWG outperforms the throughputs of the other two gateways. Figs. 17–20 show that the content request delay of our SWG is also shorter than the delays of the other two gateways. Since our SWG has a smaller LBM compared to the other two gateways, it results in our SWG having higher throughput and shorter delay.

5. Conclusions and future work

5.1. Concluding remarks

Clustering is a suitable technique for constructing a scalable, reliable, and high-performance WAP gateway. We have presented a scalable WAP gateway (SWG) which is a clustered gateway. The SWG integrates a WAP dispatcher which distributes mobile clients' requests to real gateways in kernel space and does not handle outgoing gateway-to-client responses. The WAP dispatcher realizes our efficient load balancing strategy. In our WAP-aware strategy, we estimate the potential load of real gateways in order to allocate new connection requests among the real gateways and apply an asynchronous alarm approach to provide feedback information of the actual load of real gateways. Experimental results have shown that our SWG has better load balancing performance, throughput, and delay compared to the LVS and the Kannel gateway.

5.2. Future work

Our efficient load balancing strategy can be integrated with a dynamic timeout assignment scheme which allows the WAP dispatcher to disconnect idle connections actively when the potential load of the corresponding real gateways is high. This approach can release real gateway resources and prevent potential imbalance. Although the real gateways of our SWG are fault tolerant, the fault tolerance of the WAP dispatcher deserves to further study.

Acknowledgements

This work was supported by the MOE program of Excellence under Grant 89-E-FA04-1-4 and NCTU EECS-MediaTek Research Centre under Grant Q583.

References

[1] WAP Forum, http://www.wapforum.org.
[2] Ericsson WAP Gateway, http://www.ericsson.com.
[3] Open Source Kannel Project, http://www.kannel.org.
[4] Wireless Session Protocol Specification, http://www.wapforum.org/what/technical.htm.
[5] G. Goldszmidt, G. Hunt, NetDispatcher: a TCP Connection Router, IBM Research Technical Report, RC 20853, July 1997.
[6] Linux Virtual Server Project, http://www.linuxvirtualserver.org.
[7] Linux Online, http://www.linux.org.
[8] V. Cardellini, M. Colajanni, P.S. Yu, Efficient State Estimators for Load Control Policies in Scalable Web Server Clusters, Proceedings of COMPSAC '98, 1998 pp. 449–455.
[9] K. Shen, T. Yang, L. Chu, Cluster Load Balancing for Fine-Grain Network Services, Proceedings of IPDPS, April, 2002 pp. 493–500.
[10] B.A. Julstrom, D.H. Robinson, Simulating Exponential Normalization with Weighted k-Tournaments, Proceedings of 2000 Congress on Evolutionary Computation, vol. 1, 2000 pp. 227–231.
[11] V. Cardellini, M. Colajanni, P.S. Yu, Redirection Algorithms for Load Sharing in Distributed Web-server Systems, Proceedings of 19th IEEE International Conference on Distributed Computing Systems, June, 1999, pp. 528–535.
[12] S. Floyd, V. Paxson, Difficulties in Simulating the Internet, IEEE/ACM Transactions on Networking 9 (2001) 392–403.
[13] V. Cardellini, P.S. Yu, Collaborative Proxy System for Distributed Web Content Transcoding, Proceedings of 9th International Conference on Information Knowledge Management CIKM 2000, November, 2000, pp. 520–527.
[14] J.E. Pitkow, Summary of WWW Characterizations, http://decweb.ethz.ch/WWW7/1877/com1877.htm.
[15] P. Barford, M.E. Crovella, A Performance Evaluation of HyperText Transfer Protocols, Proceedings of ACM Sigmetrics 1999, May, 1999, pp. 188–197.
[16] L. Breslau, P. Cao, S. Shenker, Web Caching and Zipf-like Distributions: Evidence and Implications, Proceedings of IEEE INFOCOM, vol. 1, 1999, pp. 126–134.
[17] R.B. Bunt, D.L. Eager, G.M. Oster, C.L. Williamson, Achieving Load Balance and Effective Caching in Clustered Web Servers, Proceedings of 4th International Web Caching Workshop, April, 1999.
[18] J.J. Huang, M.S. Chen, H.P. Hung, A QoS-Aware Transcoding Proxy Using On-demand Data Broadcasting, Proceedings of IEEE INFOCOM, March, 2004.