Contents

Abstract

Session Initiation Protocol (SIP) is the signaling protocol for VoIP. When the load on a SIP server is higher than what the server can handle for an extended period of time, the server is overloaded. This thesis implements and evaluates single and simplified multi class overload controls. The server and overload control code was instrumented to learn the costs of various operations, and they indicated that overload controls could significantly reduce server load by dropping a fraction of incoming packets. Single class overload controls treat all incoming packets equally and can reduce the latency from when call setup is initiated until the call is fully set up if they suppress retransmission. Simplified multi class overload controls separate packets into two classes: INVITE messages and other messages. Simplified multi class overload controls can increase the call throughput at the server without retransmission suppression or dramatically reduce the call latency with retransmission suppression.

1 Introduction

The goal of this thesis is to implement and investigate the advantages of single and multi class overload controls for a Session Initiation Protocol (SIP) server. SIP is a signaling protocol used for many purposes. This thesis will focus on its main use, call setup in Voice over IP (VoIP) networks. SIP is used to establish the call, control when features (e.g. call waiting) are used, and tear down the call upon completion.

Load on the SIP server varies, and when the server cannot service packets at a rate greater than or equal to the rate at which they arrive for an extended period of time, the server is said to be *overloaded*. If the arrival rate is greater than the service rate for a short period of time, then it is considered transient load, not overload. Buffers are designed to handle transient load. While a server is overloaded, if its buffer becomes full, then all incoming packets are dropped, and the queue is said to have overflowed.

A goal of overload controls is to detect overload and degrade performance gracefully. This should happen before overflow occurs and packets are dropped from the end of the queue. Decreasing call latency is another goal, as well as increasing the calls per second. Overload controls can be either single class or multi class. Single class overload controls treat all packets equally. Multi class overload controls differentiate between different packet classes, giving some preference.

The major research questions this thesis aims to answer are:

- What are the processing costs for fully processing and dropping a packet?
- What are the processing costs for overload controls including measuring system variables related to overload controls?
- What is the best method for detecting overload?
- How can this method be implemented?
- What is the best way to mitigate overload?
- How can this be implemented?
- What are the gains, if any, from single class overload controls, especially with regard to the aforementioned goals?
- What are the gains from multi-class overload controls?
- And, what are the drawbacks of overload controls?

A working implementation of overload controls was developed and integrated with a SIP server. The server was then subjected to overload. Data was gathered and analyzed to determine the processing costs and effects of overload control on the server. Specifically, the effects on calls per second and call latency were analyzed. *Calls per second* (CPS) is defined as the number of calls set up in a one second interval. *Call latency* is defined as the amount of time between when the first message to set up a call is sent and the last message is received.

The results show that dropping a packet saves a large amount of processing time and overload controls have moderately low overhead, indicating they can reduce load on the system. When measurements are done every 125 packets overhead is negligent and overload controls are still reactive.

Simplified multi class controls can increase CPS while minimally increasing call latency. Single class and simplified multi class overload controls with the ability to suppress retransmissions can significantly reduce latency while insignificantly reducing CPS.

The rest of the thesis is organized as follows: Chapter 2 gives the necessary background knowledge; Chapter 3 details the overload control algorithms; Chapter 4 details the implementation; Chapter 5 gives expected results; Chapter 6 describes the experiments and presents the results; Chapter 7 discusses related work; Chapter 8 concludes the thesis; and Chapter 9 gives acknowledgments.

2 Background

2.1 Basics

When packets come in from a network device, the operating system stores them in a queue. The packets remain there until the application they are destined for retrieves them. Because the queue is of finite size, it can become full. When the queue is full and a new packet comes in for that queue the operating system drops the packet. Then the queue is said to have *overflowed*.

2.2 VoIP

In a traditional telephony network, there are actually two networks. One network is called the data plane and is used to carry the voice data. The data plane is circuit switched. The other network is called the control plane or signaling network and is a packet-switched network. The signaling network is used to set up the circuits in the data plane for a call, as well as for call authentication, accounting, and other features.

In a Voice over Internet Protocol (VoIP) network, there are no circuits that need to be set up; however, there is still a need for the control plane. Users still must be located and authenticated, their calls tracked for billing purposes, calls still must be setup, and etc. The control plane and data plane both use the same network in VoIP. In VoIP, the signaling is accomplished using the Session Initiation Protocol (SIP) [10]. According to the SIP rfc:

> SIP invitations [are] used to create sessions carry session descriptions that allow participants to agree on a set of compatible media types. SIP makes use of elements called proxy servers to help route requests to the user's current location, authenticate and authorize users for services, implement provider call-routing policies, and provide features to users. SIP also provides a registration function that allows users to upload their current locations for use by proxy servers. [10]

To demonstrate how SIP is used we will consider two examples: how a user registers their location with a proxy server and how a call is set up.

A user registers their location with a proxy server so that he or she can receive calls at their current location. If the server does not require authentication, then registration involves two messages. The user sends a REGISTER packet to the server, and the server replies with a 200 OK packet. If the server requires authentication, registration involves four messages. The user sends a regular REGISTER packet to the server, the server replies with a

407 Proxy Authentication Required packet, and then the user then resends the REGISTER packet with authentication information (e.g. their password). After authenticating the user, the server then responds with a 200 OK packet. Figure 1 shows this exchange.

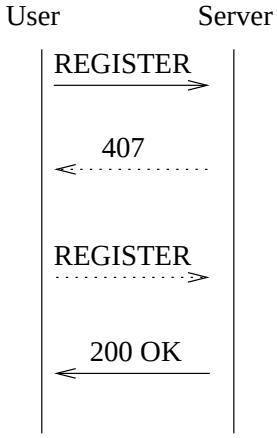

Figure 1: **Illustration of a user registering with a server. A non authenticated register includes only the solid line messages; an authenticated register includes both solid and dotted line messages.**

To detail how a call is set up, consider a sender who wants to call a receiver. For simplicity, we will assume the sender and the receiver have both registered with the same server. The sender sends an INVITE message to the server. The server then forwards the message to the receiver and sends the sender a 100 Trying message. When the receiver's phone receives the INVITE, it immediately responds with a 180 Ringing message back to the server, which then forwards it to the sender. When the receiver answers his phone, it sends a 200 OK message to the server, which forwards it to the sender. The sender then sends an ACK directly to the receiver, and their conversation starts using a different protocol. RTP [4] is usually used to carry the voice data. At the end of the conversation, one phone sends a BYE message directly to the other, who then responds with a 200 OK message. In a VoIP setting, the ACK and BYE messages are sent through the server. This allows for features such as call accounting. Figure 2 shows the INVITE message exchange.

2.3 Classes

Overload controls can be either single class, multi class, or simplified multi class. This thesis implements and evaluates both single and simplified multi class overload controls. The following subsections explain single class, multi class, and simplified multi class overload controls. All of the overload controls give a percentage of packets to be accepted. By accepting only a certain percentage of packets, the overload controls can decrease call setup

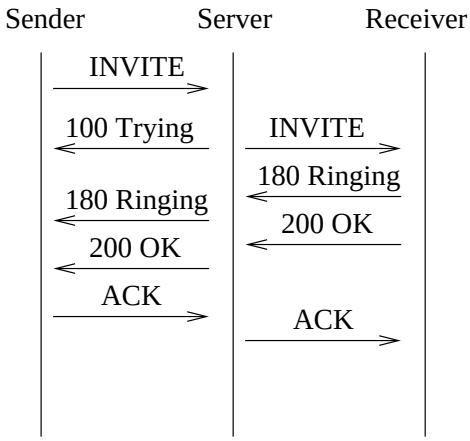

Figure 2: **Illustration of a sender initiating a call with a receiver. It is assumed that both the sender and receiver have registered with the server before this.**

latency and give certain classes priority over others.

2.3.1 Single Class

Single class overload controls do not differentiate between different types of packets. INVITEs, REGISTERs, ACKs and all other packets are treated equally. Single class overload controls are simpler to implement than multi class overload controls and require less setup than multi class overload controls. The main advantage of single class overload controls is that they can reduce the delay before incoming packets are serviced, thus reducing call latency; however, this is at the expense of the overall accepted packet processing rate. Single class overload controls calculate a fraction of the incoming packets to allow.

2.3.2 Multi Class

Multi class overload control differentiate between different types of packets. By dropping certain types of packets, they can give preference to others. For instance, if a server dropped every message it received that was not a REGISTER, it could service many more REGISTER messages. Or, a server could service more packets of the general type if it dropped all non essential packets (e.g. 180 Ringing). Multi class overload controls calculate a fraction of incoming packets to allow for each class.

2.3.3 Simplified Multi Class

Simplified multi class overload controls have only two classes, INVITES and all other packets. This is based on the fact that every INVITE that is accepted will lead to at least a 180

Ringing, 200 OK, ACK, BYE, and 200 OK packets. Thus, dropping an INVITE implicitly saves a lot of future traffic. Also, once a phone starts ringing, it would be inappropriate for that call not to be fully set up or, after the call is over, to not be allowed to hang up. Simplified multi class overload controls give a fraction allowed for INVITEs; the fraction allowed for all other classes is always one.

2.4 Dropping Level

Dropping can be done either at the application level or the kernel level. The kernel processing is earlier in the processing path for packets than application level processing, so dropping in the kernel will save more processing time. Dropping packet at the kernel level, however, is more difficult to implement and less portable. Dropping packets in the kernel is part of future work.

3 Algorithms

Applying overload controls consists of mainly two steps. The first step is identifying the overload, while the second is attempting to mitigate its effect. The overload detection algorithms are very similar to those described in [6] and [7]. All the overload controls are event driven, meaning they are invoked whenever a packet arrives, as opposed to time driven. Each of the overload control algorithms gives a fraction allowed, f. This is the fraction of incoming packets to be accepted and is directly applied in the single class case. The fraction allowed is applied directly to invite messages in the simplified multi class case.

3.1 Occupancy

Processor occupancy, ρ, is the percentage of total time spent by the processor on the SIP server's proccesses. Note that this definition differs from [7] in that it only takes the SIP server's proccesses into account as opposed to the entire load on the system. In an environment where a SIP server is deployed, it will be the main function of the machine and other tasks such as OS background tasks should have little impact. In addition, measuring the system load presented by the other tasks has prohibitive processing costs. Thus, ρ in this thesis effectively approximates ρ in [7]. The implementation section describes precisely how ρ is measured. In short, the last k values are used to approximate the instantaneous processor occupancy.

$$\rho_n = \frac{systemtime_n - systemtime_{n-k}}{clocktime_n - clocktime_{n-k}} \tag{1}$$

$systemtime_n$ is the scheduled system time for the SIP server's processes. An exponentially weighted moving average (EWMA) is also used with new value weight w on ρ to obtain an estimate of recent processor occupancy, $\hat{\rho}$.

$$\hat{\rho}_n = (1 - w)\hat{\rho}_{n-1} + w\rho_n \tag{2}$$

Using an EWMA prevents the overload controls from unnecessarily reacting to transient load. The estimate, $\hat{\rho}$ is then compared to ρ_{thresh}. The system detects overload whenever $\hat{\rho}$ exceeds ρ_{thresh}. The fraction allowed, f, is varied by the following equation.

$$f_{n+1} = restrict(f_{min}, \frac{\rho_{thresh}}{\hat{\rho}} f_n, 1) \tag{3}$$

Restrict(a, x, b) = max(a, min(x,b)), and f_{min} is a minimum fraction allowed. Because f

can never be lower than f_{min}, at least f_{min} percent of packets will be fully processed. This is necessary to prevent the system from dropping every packet when it is severely overloaded.

3.2 Acceptance Rate

A drawback of using only processor occupancy to detect overload is that after a certain point, it cannot indicate to what degree a system is overloaded. If a system has 100% processor occupancy, the offered load might be 100% of what the machine can handle or it may be 200% of what the machine can handle. This problem motivates the use of acceptance rate in combination with processor occupancy to detect overload. In the single class case, acceptance rate is identical to the occupancy algorithm, except it uses call acceptance rate α instead of processor occupancy ρ and does not use an EWMA. ARO finds the percentage difference between offered and accepted packets out of the last $(k-1) * measure$ packets.

$$\alpha_n = \frac{acceptedpackets_n - acceptedpackets_{n-k}}{totalpackets_n - totalpackets_{n-k}} \tag{4}$$

Acceptance rate occupancy (ARO) is a combination of acceptance rate and processor occupancy. The fraction allowed is given by the following equation:

$$f_{n+1} = restrict(f_{min}, min(f_\alpha, f_\rho), 1) \tag{5}$$

$f_\alpha = f$ from the acceptance rate algorithm and $f_\rho = f$ from the processor occupancy algorithm.

3.3 SiRED

SiRED is a derivative of Random Early Detection (RED)[3]. SiRED takes a measurement of queue length and divides it by the maximum size of the queue to get queue occupancy, q. An EWMA with new value weight w is used to obtain an average queue length.

$$Q_{n-1} = (1-w)Q_n + wq \tag{6}$$

SiRED has two threshold values: Q_{min} and Q_{max}. Q_{min} is the threshold above which the system detects overload. Q_{max} is the threshold above which the system detects severe overload, and the most serious measures are taken to avoid overflow. An EWMA is again

used with SiRED to smooth its behavior.

$$\hat{q} = (1 - w)\hat{q}_{n-1} + wq \tag{7}$$

The fraction allowed is given by the following equation:

$$f = restrict(f_{min}, \frac{Q_{max} - \hat{q}}{Q_{max} - Q_{min}}, 1) \tag{8}$$

All packets are allowed when the queue occupancy is less than Q_{min}, and when it is greater than Q_{max}, only f_{min} packets are allowed. When queue occupancy is between these values, it is given by the linearly decreasing function $\frac{Q_{max}-\hat{q}}{Q_{max}-Q_{min}}$.

3.4 Dropping

The only way to mitigate overload is to drop a fraction of the incoming packets. Dropping packets requires less processing time than accepting them, so more packets can be processed per unit time, which reduces the overload. Each of the preceding algorithms gives f – the fraction of packets to allow. For example, if $f = .5$, then half of all packets should be accepted and half should be dropped.

The question then becomes how to accept the fraction f of packets. A simple solution would be to generate a random number in the range [0,1] and accept the packet if the random number is less than or equal to f. The drawbacks of this approach are twofold. First, because random numbers are used, the actual fraction allowed would deviate from the desired f value. Second, according to [6], this approach produces clusters of throttled calls. This is turn leads to synchronization problems.

A more elegant solution is proposed by [3]. It is a deterministic scheme that evenly distributes the dropped calls. Let r initially = 0.

```
r := r + f
if r ≥ 1 then
   r := r - 1
   accept call
else
   reject call
end if
```

For example, if $f = \frac{1}{3}$, then every third call will be accepted.

4 Implementation

Implementing the overload controls involved two steps. The first was obtaining, setting up, and implementing the overload controls on a SIP server. The second was generating the load to test the SIP server.

4.1 SER

The SIP Express Router (SER) [11] was chosen as the SIP server. The main reasons for choosing it were that it is open source and that it is commonly used.

Obtaining SER was technically trivial. It simply involved downloading the source code and compiling it. The version used was 0.9.6.

SER uses configuration files that define the modules to use, the parameters to pass the modules, and a script to run whenever a packet is received. Setup involved modifying the configuration files.

4.1.1 Overload Controls

The overload controls were initially implemented as stand-alone C programs for initial testing purposes. For the various overload control algorithms, the following needed to be measured: processor occupancy, queue occupancy, and arrival rate.

Processor occupancy was initially measured by noting the current number of user and system jiffies that the process has been scheduled for and comparing it with the previous values. Jiffies are operating system defined units of time. The number of user and system jiffies used were obtained from the utime and stime fields in the /proc/"pid"/stat file, which keeps various statistics about currently running processes; however, in the version of Linux used jiffies were 1/100ths of a second. Because packets were received at rates much higher than 100 per second, the processor occupancy measurements varied wildly. Most processor occupancy readings were 0% and a few were in the 3000% to 4000% range. This is clearly inaccurate, so the time window for comparison was increased. A module parameter k is used and can be specified in the configuration file used by SER. The k parameter indicates the number of previous measurements to keep. The current measurement is compared against the oldest measurement to obtain the processor occupancy. This is efficiently implemented in a singly linked list. Also, taking a measurement every $measure$ number of packets increases the time window. To be precise, the time window is equal to the time it takes for $(k-1)*measure$ packets to be processed. The values for k and $measure$ used in the experiments are large

enough that processor occupancy values are representative and correlate with values obtained from top, a Linux command line tool.

Queue occupancy was obtained the /proc/net/udp file. The /proc/net/udp file is the UDP socket table and contains, among other information, the current size of the receive queue for SER. Because that file is a dynamically sized table, the position of the receive queue size is not known ahead of time. The solution is to search for the port number of SER, 5060 by default. Then, given the port position, extract the receive queue size. The overload controls need queue occupancy as a percentage, so the receive queue size was divided by the maximum size of the queue. The maximum size of the queue was experimentally determined.

Arrival rate was much simpler to implement. Arrival rate needs the number of packets offered and the number of packets fully processed. Both of these are application-level statistics, so determining arrival rate requires keeping two counters and a single division operation. Similar to processor occupancy, arrival rate also compares the current measurement to a previous measurement. Thus, module parameters k and $measure$ are used in arrival rate in the same way they are used in processor occupancy.

SER uses modules to provide additional functions to its core. SER modules are implemented in C so there was a direct translation of the stand alone control programs to modules. The parameters for the overload controls are defined in the configuration files for SER and passed to the modules on start up. The script in the configuration file calls the module and then uses its return value to determine if the packet should be accepted or dropped.

4.2 SIPp

SIPp [12], an open source SIP packet generator, was downloaded, installed, and used to load the server. The specific version of SIPp used was the unstable source for version 1.1 dated January 8, 2007. The unstable source was used instead of the stable source because of its increased call load generation ability. There were numerous challenges presented by SIPp.

SIPp instances are only allowed to send packets or receive and respond to packets, but not both. In the SIP rfc [10], each device is expected to have a user agent client (UAC) and a user agent server (UAS). The UAC sends packets (e.g. initiates calls with INVITE messages), while the UAS receives and responds to packets (e.g. receiving and responding to an INVITE message). SIPp instances cannot be both a UAC and a UAS. This presents problems because in the experiments the senders and receivers both have to register with the server and then proceed with setting up and tearing down calls. Thus, the receiver had to have both a UAC and a UAS. The solution was a shell script that started one SIPp instance,

had it register with the server, and then started a new SIPp instance to send or receive calls. Note that this was not strictly necessary for the senders, as they are UACs the whole time; however, it made setting up the experiments easier and cleaner.

The other major challenge presented by SIPp was that none of the available machines were able to generate enough load to overload the server on their own. This was handled by using four computers to generate the load. How they interacted and were controlled is detailed in the following subsection.

4.3 Coordination

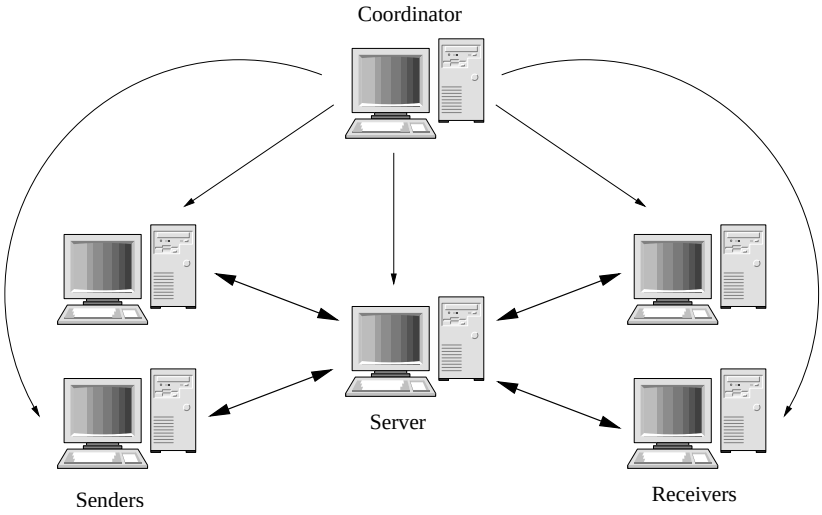

Figure 3: **The experimental setup. The bold lines indicate SIP traffic, the regular lines indicate coordination information.**

Overload was created on the server by using two pairs of computers to set up and tear down calls. In each pair, there was a sender and a receiver. The sender initiated both call setup and tear down, and the receiver responded positively to all the messages it received. The server was an intermediary in all of the exchanges. In addition, another computer – called the coordinator – was used to coordinate all four computers to generate overload. The coordinator also communicated with the server to allow automation of the experiments. Figure 3 shows the setup. The following table shows the output from the tcpdumprogram from the server (eagles) for a call setup and tear down between a sender (schuylkill) and a receiver (mets). Each line of output from tcpdump is followed by a # and a comment on which packet it is in the sequence. This demonstrates that all the packets were being sent through the server.

```
IP schuylkill.cse.psu.edu.5060 > eagles.cse.psu.edu.5060: UDP, length 338 #INVITE → server
IP eagles.cse.psu.edu.5060 > schuylkill.cse.psu.edu.5060: UDP, length 515 #100 Trying → sender
IP eagles.cse.psu.edu.5060 > mets.cse.psu.edu.5060: UDP, length 448       #INVITE → receiver
IP mets.cse.psu.edu.5060 > eagles.cse.psu.edu.5060: UDP, length 351       #180 Ringing → server
IP mets.cse.psu.edu.5060 > eagles.cse.psu.edu.5060: UDP, length 521       #200 OK → server
IP eagles.cse.psu.edu.5060 > schuylkill.cse.psu.edu.5060: UDP, length 289 #180 Ringing → sender
IP eagles.cse.psu.edu.5060 > schuylkill.cse.psu.edu.5060: UDP, length 459 #200 OK → sender
IP schuylkill.cse.psu.edu.5060 > eagles.cse.psu.edu.5060: UDP, length 331 #ACK → server
IP schuylkill.cse.psu.edu.5060 > eagles.cse.psu.edu.5060: UDP, length 338 #BYE → server
IP eagles.cse.psu.edu.5060 > mets.cse.psu.edu.5060: UDP, length 420       #ACK → receiver
IP eagles.cse.psu.edu.5060 > mets.cse.psu.edu.5060: UDP, length 448       #BYE → receiver
IP mets.cse.psu.edu.5060 > eagles.cse.psu.edu.5060: UDP, length 347       #200 OK → server
IP eagles.cse.psu.edu.5060 > schuylkill.cse.psu.edu.5060: UDP, length 285 #200 OK → sender
```

The coordination was done using shell scripts and simple socket programs written in C. The server ran a shell script called runser.bash that alternated between two programs. The first program opened a socket and waited for a UDP packet, which contained the configuration file to use. When the packet arrived, the program would print the configuration file. The shell script then ran SER with that configuration file. The following shows a simplified version of runser.bash:

```
#!/bin/bash
#runser.bash
while [1]; do
  #Listen for the name of the configuration file
  config=$(./waitforsippsignal 30045)

  #kill previous instance of SER
  ser_pid=$(ps | grep ser | grep -v run | awk -F" " '{ print $1 }' | head -1)
  if [ "$ser_pid" != "" ]; then
    kill -9 "$ser_pid"
  end if

  #If the message is done then quit
  if [ "$config" = "done" ]; then
    exit 0
  end if

  #Run ser with the given config file
  ./ser -l eagles -m 1024 -f ~/Thesis/SER/configs/"$config".cfg &
end while
```

Each of the four computers involved in overloading the server ran a script similar to the one the server ran, called follower.bash. The only difference was that these scripts received the name of another shell script in the packet, and then ran that shell script. The following is a simplified version of follower.bash:

```
#!/bin/bash
```

```
#follower.bash
while [ 1 ] do
  #Listen for the name of the script to run
  script=$(./listenforleader 30045)

  #If the message is done then quit
  if [ "$script" = "done" ]; then
    exit 0
  end if

  ./"$script"".bash"
end while
```

The second script was defined by the different experiments being run, but they all had a similar structure. The script would first start a SIPp instance that would register that computer with the server. Then it would start the main SIPp instance. For receivers, the second script would simply listen and respond to messages. For senders, the second script would start the SIPp instance and then tell it the rate at which to send via a local UDP port. Then, at the completion of the experiment, the script would gather the statistics files generated by the main SIPp instance, name it according to its role (e.g. sender1), and move it to a predefined location so the coordinator could handle it. The following shows a simplified version of grad_s1.bash, the second script used for the first sender in the gradual overload experiment:

```
#!/bin/bash
#grad_s1.bash

#Register with the server
./sipp -m 1 -bg -sf ../scenarios/uac_1-1.xml eagles

#Allow the registration to complete, and the receiver's registration to complete
sleep 5

#Start the sender, but don't have it start inviting yet
./sipp -bg -trace_stat -fd 1 -r 0 -sf ../scenarios/uac_1-2.xml eagles

#Get the pid of sipp so we can kill it and find its stat file later
sipp_pid=$(ps | grep sipp | head -1 — awk '{ print $1 }')

#wait 10 seconds then increase the call load by 100 every second for 50 seconds
#then wait 30 seconds and decrease the call load by 100 every second for 50 seconds
sleep 10
for ((i=0;i<500;i++)); do
  sleep .1
  echo "*" >/dev/udp/127.0.0.1/8888
end for
sleep 30
for ((i=0;i<500;i++)); do
  sleep .1
```

```
  echo "/" >/dev/udp/127.0.0.1/8888
end for
sleep 15

#The trial is over, so kill the SIPp instance
kill -9 "$sipp_pid"

#Find the statistics file, put it in a central location for the leader, and rename it sender1.csv
cd ../scenarios
file=$(ls *_"$csv_pid"_*)

mv "$file" "$dest_dir""$csv_name"".csv"
```

The coordinator ran a more complex script called leader.bash. For each experiment, the coordinator had the name of the scripts for each of the four computers that were overloading the server and the configuration file for the server. The coordinator then sent the file names to the other computers. Next, it waited for all of the statistics files to appear in a predefined location. Once all the statical files appeared, it renamed them according to experiment and run number. Then, it started on the next run by sending the names of the scripts to use to the four overloading computers and the configuration file to the server.

4.4 Machine Details

This section gives the exact details of the machines used for each of the roles described previously. All the information was gathered from the /proc/cupinfo, /proc/meminfo, /etc/fedora-release, or /etc/redhat-release files, or from the uname command for the Linux machines. The information on the SunOS machine was gathered with the psrinfo, uname, and top commands.

The server ran on a Intel Pentium 4 CPU running at 3.40 GHz. The cache was 1024 KB and main memory was 1GB. The OS was Fedora Core 4 with the 2.6.20 Linux kernel.

The coordinator ran on a Sun-Fire-880 with 8 processors running at 900 MHz. The main memory was 28 GB. The OS was SunOS 5.9 with a Generic_118558-39 kernel. This was a multi user machine; however, the coordinator did not require much processing power, so this was not an issue.

The first sender ran on a machine with four Intel Xeon CPUs at 2.80 GHz. Each of the four processors had a 512 KB cache and the machine had 2 GB of main memory. The OS was Red Hat Enterprise Linux AS release 3 (Taroon Update 8) with a 2.4.21-32.ELsmp Linux kernel. This was also a multi user machine, but it experienced infrequent, light use by other students, so it was not an issue.

The second sender ran an Intel Pentium 4 CPU running at 3.40 GHz. The cache was 1024

KB and main memory was 1GB. The OS was Fedora Core 4 with the 2.6.11-1.11369_FC4smp Linux kernel.

The first receiver ran an Intel Pentium 4 CPU running at 3.06 GHz. The cache was 512 KB and main memory was 1GB. The OS was Fedora Core 4 with the 2.6.14-1.1656_FC4 Linux kernel.

The second receiver ran an Intel Pentium 4 CPU running at 3.06 GHz. The cache was 512 KB and main memory was 2GB. The OS was Fedora Core 5 with the 2.6.15-1.2054_FC5smp Linux kernel.

5 Expected Results

This section will discuss the expected results for the experiments in the next section.

5.1 Packet Processing Rate

The packet processing rate, PP_r, is the number of packets that can be processed per second. The accepted packet service rate, PP'_r, is the number of accepted packets that are processed per second. The accepted packet service rate is very important in terms of the server performance. It directly relates to calls per second (CPS), which is one of the metrics for evaluation. CPS is the number of calls successfully set up per second. Each successful call setup includes at least an INVITE message and a 200 OK message, and should include a 100 Trying and a 180 Ringing message. One of the goals of overload controls is to maximize CPS and thus PP'_r. Recall that fraction allowed f, is the percentage of packet to be fully processed. Then $PP'_r = PP_r * f$. Let t_a be the time to process an accepted packet, and let t_d be the time to process a dropped packet. Then PP_r and PP'_r are:

$$PP_r = \frac{1}{t_a * f + t_d * (1 - f)} \tag{9}$$

$$PP'_r = \frac{f}{t_a * f + t_d * (1 - f)} \tag{10}$$

Because PP'_r is a strictly increasing function, it is maximized when $f = 1$. Overload controls can only increase t_a and t_d compared to no overload controls. Also, no overload controls implicitly set $f = 1$. Thus, PP'_r is maximized with no overload controls; however, when $t_a \gg t_d$, there is a large range of f values with high throughput. There is an example at the end of the section to illustrate this point.

5.2 Call Setup Latency

Call setup latency, lat, is the amount of time between a sent INVITE and a received 200 OK message. Call setup latency involves multiple packets, so for simplicity consider only a single packet. The time to physically transmit the packet, process it at the server, let it wait in the destination queue, and process it at its destination is assumed to be an average, $\bar{a}$, for all settings. Then $lat_{packet} = \bar{a} + time_q$, where $time_q$ is the time spent in the queue at the server. Thus the latency of a packet is directly proportional to the time it spends in the queue at the server. $time_q = \frac{q_{packets}}{PP_r}$, where $q_{packets}$ is the number of packets in the queue. So, in the worst case, $lat = \frac{q_{full}}{PP_r}$ and in the normal case $lat = \frac{q_{avg}}{PP_r}$, where q_{full} is the size of

the queue in packets and q_{avg} is the average number of packets in the queue. Thus, average latency can be decreased by increasing PP_r and/or decreasing q_{avg} and worst case latency can be decreased by increasing PP_r and/or decreasing q.

If you assume that overload controls introduce no overhead, then whenever they drop packets, they increase PP_r. This will in turn also reduce q_{avg}. Thus, overload controls will reduce call setup latency and having no overload controls maximizes latency. However, if overload controls introduce some overhead, which they do, this may not necessarily be true. If the overhead from the overload controls is greater than the savings from dropping packets, then latency will actually increase. Therefore, it is important for overload controls to have low overhead.

Retransmissions also present an interesting problem when dealing with call set up latency. If an INVITE is dropped, and then its retransmission is accepted, when should the timer for call setup latency start? Should it be when the initial INVITE is sent or when the retransmitted INVITE is sent? Intuitively, call set up latency is experienced by the user (i.e. a person), so they are only aware of when the first INVITE is sent. This indicates that overload controls could actually significantly increase call latency if they accept a retransmission of an earlier dropped INVITE. The solution is to stop INVITES from being retransmitted. The 200 OK response could also be retransmitted multiple times, which would also increase call latency.

There is a trade-off between call setup latency and CPS. As CPS increases, so does latency, and as latency decreases, so does CPS. The question then becomes: What is more important? This is somewhat subjective, but ideally there would be very high CPS with very low latency. The relationship between CPS and latency is, however, not linear, and therefore single class overload controls can have some tangible benefit. CPS directly correlates with PP'_r, and call latency directly correlates to f. Figure 4 shows the relationship between PP'_r and f, and the next section gives an example.

5.2.1 Example

Consider the following example based on values from the experimentation section. If CPS decreases by 10% how much will call latency decrease? Let $t_a = 70\mu sec$ and $t_b = 6\mu sec$. Then $PP'_{r_{max}} = \frac{f}{64f+6} * 10^6 = 14285.7$ CPS from equation10 with $f = 1$. Take 90% of this and $PP'_r = 12857.1$, and equation10 gives f $\approx .44$. Now by equation 9 $PP_r \approx 32000$. Thus *lat* caused by waiting in the queue decreases by at least a factor of 2.2, with only a 10% reduction in PP'_r.

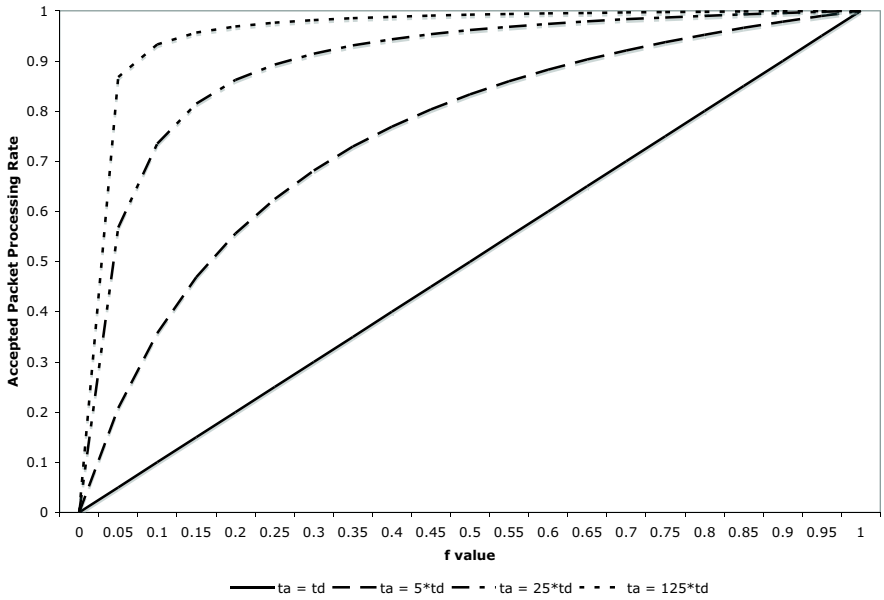

Figure 4: **Relationship between PP'_r and f for different relationships between t_a and t_d**

6 Experiments

This section details the various experiments run, their motivation, and results.

6.1 Code Instrumentation

In this experiment, the code was instrumented. The motivation was to find the processing costs of different types of packets. In addition, the processing cost was found for overload controls in general, overload controls with a measurement taken, and dropping a packet.

The results were gathered for two intervals. The first is the full cycle interval, which is the time difference between receiving a packet at the application level and receiving the packet that immediately follows it at the application level. The full cycle interval represents all time spent processing the packet at both the user and system level. The second interval is the packet processing (pack proc) interval, which is the time spent processing the packet from when the application receives the packet until it has fully processed it. It does not include the kernel time spent to receive or send the packet. The measurements were made using the rdtsc instruction [9]. The rdtsc instruction returns a 64 bit value from the processor time stamp counter, which is incremented every clock tick. The resulting values are then divided by the processor's clock frequency to get the time in seconds.

The experiments consisted of 10 runs, and each run was 100,000 packets long. The minimum, median, and mean values were collected for each run. The statistics shown here are the minimum of the minimums, the median of the medians, and the mean of the means.

The overload column gives the overload control used. The values for all overload controls were very similar so the values for processor occupancy are shown. *Basic* is the baseline system with no overload controls. The packet column gives the type of packet. The *good/bad* column indicates if a packet is addressed to a registered user (good) or if a packet is addressed to a non existent user (bad). To detect overload, measurements of queue occupancy, processor occupancy, and/or arrival rate must be taken. These measurements are done every K packets. The *measure* column indicates whether a measurement was taken. The *accept/drop* column indicates if the packet was dropped or accepted. The minimum, median, and mean values are given in microseconds.

The experiment was performed by overloading the server with the specified packet type, and by configuring the server to either always measure or never measure and to always accept or to never accept.

config	packet	good/ bad	measure measure	accept/ drop	full cycle min	full cycle median	full cycle mean
basic	ack	bad	-	-	15.81	87.49	173.8
occupancy	ack	bad	no	accept	16.07	90.88	177.69
occupancy	ack	bad	yes	accept	16.81	106.71	181.6
occupancy	ack	bad	no	drop	2.76	78.67	180.28
occupancy	ack	bad	yes	drop	5.21	92.59	329
basic	bye	bad	-	-	16	71.02	174.38
occupancy	bye	bad	no	accept	16.23	48.12	173.42
occupancy	bye	bad	yes	accept	16.8	106.9	352.13
occupancy	bye	bad	no	drop	2.72	6.04	196.11
occupancy	bye	bad	yes	drop	5.25	93.93	1024.89
basic	invite	bad	-	-	4.2	44.22	153.89
occupancy	invite	bad	no	accept	4.37	49.47	92.71
occupancy	invite	bad	yes	accept	4.3	17.26	758.64
occupancy	invite	bad	no	drop	2.76	5.14	122.38
occupancy	invite	bad	yes	drop	5.23	106.73	425.83
basic	ack	good	-	-	26.69	87.61	669.44
occupancy	ack	good	no	accept	27.15	45.28	681.73
occupancy	ack	good	yes	accept	27.55	121.2	671.53
occupancy	ack	good	no	drop	2.75	4.94	1864.66
occupancy	ack	good	yes	drop	5.12	92.45	1554.85
basic	bye	good	-	-	15.9	71.76	175
occupancy	bye	good	no	accept	16.11	59.46	173.64
occupancy	bye	good	yes	accept	16.81	106.04	175.68
occupancy	bye	good	no	drop	2.75	90.18	184.85
occupancy	bye	good	yes	drop	5.18	93.21	177.72
basic	invite	good	-	-	9.68	37.37	2046.66
occupancy	invite	good	no	accept	9.74	22.36	557.32
occupancy	invite	good	yes	accept	14.83	28.56	1235.31
occupancy	invite	good	no	drop	2.77	25.45	1350.34
occupancy	invite	good	yes	drop	5.24	100.88	1401.71

Full Cycle Statistics

The minimum values for full cycles are noticeably more consistent than the median or mean. For instance, notice that dropping a bad bye message on average took significantly longer than fully processing it. Also, the median value for dropping a packet and not measuring has three values around 5 or 6 microseconds and two above 75 microseconds, though it is largely packet independent. The mean varies so much due to outliers. Outliers can be caused by context switches to other processes. In Linux time-slices are normally around 10 milliseconds, so this would have a dramatic effect on mean. The varying values for median suggest that there are a large number of packet processed at close to the minimum time and a large amount of packets processed at much higher than the minimum time. The most probable explanation for this is context switches; however, because the median is

dramatically lower than the mean this indicates that a majority of the context switches are to the kernel– probably so it can handle the IO for incoming and outgoing packets. Another possible cause of the outliers is page faults for the files being read. This is unlikely though since the files are accessed frequently by SER and the kernel. Because the mean and median value are less consistent, the rest of the analysis of this data will focus on the minimum values.

An important point to note about the full cycle statistics is that the minimum for dropped packets is consistent across all packet types. Intuitively, this makes sense because no packet type specific processing is done before the packet is dropped. When no measurement is performed, dropping a packet takes approximately $2.75\mu secs$, and when a measurement is performed it takes approximately $5.25\mu secs$. The cost for overload controls both with and without measuring is consistent across packet types. The cost for overload controls without measuring is the difference between the minimum value for basic and the minimum value for occupancy that accepted and did not measure. The cost for overload controls with measuring is the difference between the minimum value for basic and occupancy that accepted and did measure. Overload controls without measuring took approximately $.2\mu secs$ and overload controls with measuring took approximately $1\mu secs$. Depending on the packet type, overload controls without measuring adds between $<1\%$ and 5% overhead.

Some interesting conclusions can be drawn from the data are as follows. Dropping a packet, especially when not measuring, saves a lot of processing time compared to accepting it. This indicates that overload controls should be able to significantly decrease latency, while not altering CPS significantly. Another interesting point is that measuring on dropped packets is much more costly than measuring on an accepted packet. This indicates that it would be optimal to measure only for accepted packets. A good scheme would be to accept every packet that was measured. Investigating this approach is future work.

config	packet	good/ bad	measure measure	accept/ drop	full cycle min	full cycle median	full cycle mean
basic	ack	bad	-	-	14.13	15.54	16.43
occupancy	ack	bad	no	accept	14.28	15.34	16.29
occupancy	ack	bad	yes	accept	14.88	15.68	62.61
occupancy	ack	bad	no	drop	1.62	4.03	3.65
occupancy	ack	bad	yes	drop	3.83	4.18	47.45
basic	bye	bad	-	-	14.2	15.64	16.59
occupancy	bye	bad	no	accept	14.45	15.4	16.49
occupancy	bye	bad	yes	accept	14.86	15.6	59.97
occupancy	bye	bad	no	drop	1.63	3.58	3.77
occupancy	bye	bad	yes	drop	3.84	4.15	47.61
basic	invite	bad	-	-	2.94	6.23	17.06
occupancy	invite	bad	no	accept	2.97	23.61	16.89
occupancy	invite	bad	yes	accept	2.94	66.18	42.11
occupancy	invite	bad	no	drop	1.63	1.82	3.54
occupancy	invite	bad	yes	drop	3.75	4.15	48.69
basic	ack	good	-	-	24.97	26.68	29.49
occupancy	ack	good	no	accept	25.44	26.66	29.35
occupancy	ack	good	yes	accept	25.72	27.25	77.93
occupancy	ack	good	no	drop	1.65	1.84	3.73
occupancy	ack	good	yes	drop	3.8	4.15	47.55
basic	bye	good	-	-	14.21	15.57	16.62
occupancy	bye	good	no	accept	14.37	15.37	16.5
occupancy	bye	good	yes	accept	14.89	15.59	62.87
occupancy	bye	good	no	drop	1.66	4.07	3.86
occupancy	bye	good	yes	drop	3.83	4.07	47.8
basic	invite	good	-	-	7.6	47.91	38.97
occupancy	invite	good	no	accept	7.72	59.28	53.83
occupancy	invite	good	yes	accept	13.12	33.01	49.04
occupancy	invite	good	no	drop	1.66	3.72	4.12
occupancy	invite	good	yes	drop	3.76	4.21	49.02

Packet Processing Statistics

Packet processing time is shorter than full cycle time and is done in the context of a single packet. This leads to fewer outliers in this data compared to the data for the full cycles.

Dropping a packet without measuring is consistent across all the packets and takes approximately $1.65\mu secs$. The difference between this and the full cycle time is approximately $1.1\mu secs$. This difference is the kernel time spent on packets, meaning that in kernel dropping would very likely take approximately $1.1\mu secs$, which is less than half the time of application level dropping.

Dropping a packet without measuring is significantly less on average than dropping a

packet while measuring. Also, measuring has a lower overhead cost for accepted packets. This supports the previous conclusion that packets that trigger a measurement should always be accepted to save overhead.

6.2 End-to-End Overhead

This experiment was designed to test the end-to-end overhead associated with overload controls. The experimental variable was *measure*, which is the frequency for measuring in terms of packets. Setting $measure = x$ means the overload controls will measure every x packets.

The experiment tested end to end overhead by observing the CPS at the receivers. The higher the overhead then the lower the CPS will be.

The experiment varied the *measure* parameter but kept all other settings the same. The server was also configured to accept every packet, so the effect of the overload controls would not be observed, only their overhead. Initially two receivers registered with the server and started keeping statistics. Then 10 seconds later, two senders started setting up and tearing down calls using the server as an intermediary. The senders increased their call load at a constant rate of 100 calls per second for a total of 50 seconds. Each configuration was run 25 times and the averages are shown. Graph 5 shows the CPS for *measure* = powers of 10. Graph 6 shows the average CPS for *measure* = powers of 10 and 125, 250, 375, and 500.

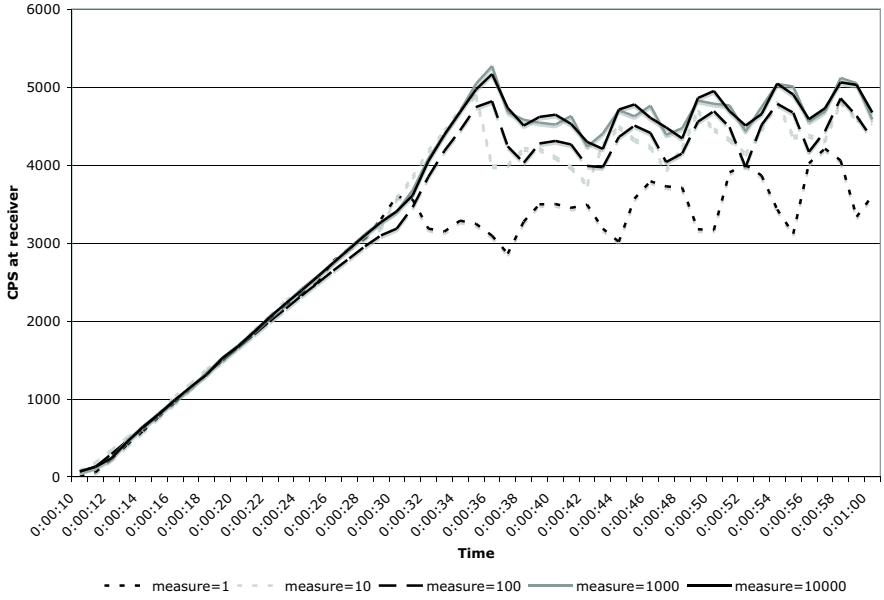

Figure 5: **End to End Overhead**

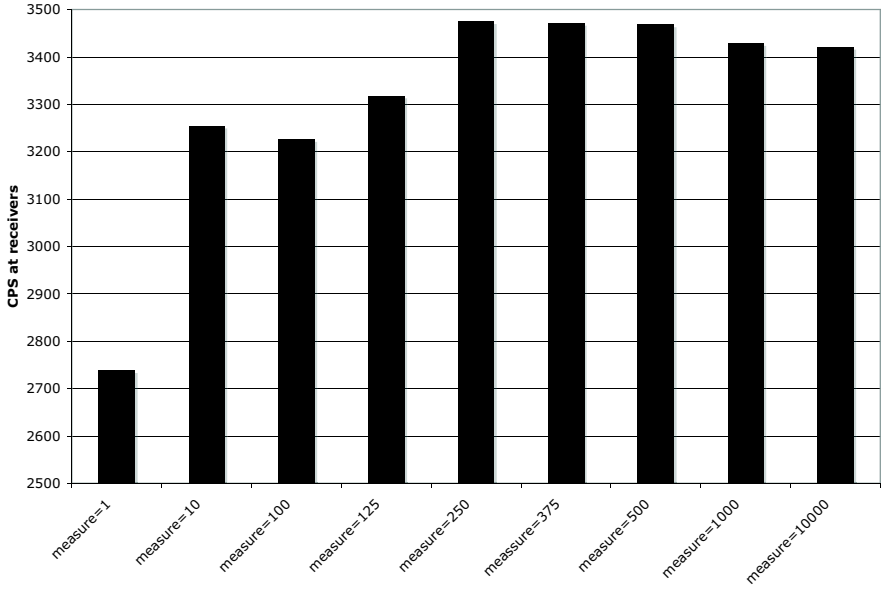

Figure 6: **Averaged End to End Overhead**

Note in the first graph that in the first 30 seconds, CPS at the receivers is essentially the same for all values of *measure*. This suggests that the overhead from overload controls does not affect CPS when the server is not overloaded.

The CPS is noticeably lower when $measure = 1$ compared to the other powers of ten. This indicates that measuring while processing every packet will noticeably degrade throughput. When *measure* is greater than 250, CPS is no longer affected. When *measure* is greater than 125, CPS is minimally affected. Thus, setting *measure* higher than 250 results in no significant overhead gains, and setting *measure* higher than 125 results in a minimal increase in CPS.

6.3 Reactiveness

Overload controls need to be able to react quickly to overload to allow graceful degradation of services. Given some system parameters and arrival rates shown in later experiments, the frequency of measurement for a given reactiveness can be mathematically determined. With no EWMA, the Equation 11 gives the maximum value for measure to guarantee that overload will be detected before the buffer overflows.

$$measure = Q_s * (1 - min) \tag{11}$$

Q_s is the size of the queue in packets, and $0 < min < 1$ is the threshold for declaring overload. Note that this is independent of the arrival and service rate of packets, indicating it could be dynamically adjusted. This dynamic adjustment is part of the future work.

With an EWMA with new weight average w, if the load is increasing at a constant rate then $f_n \approx val_{n-(\frac{1}{w}-1)}$. Thus to detect overload before overflow occurs there must be $\geq \frac{1}{w}$ measurements in the time it takes to get from min to overload. With an EWMA Equation 12 gives the maximum value for measure to guarantee that overload will be detected before the buffer overflows.

$$measure = Q_s * (1 - min) * w \tag{12}$$

If the goal is to detect overload in a short amount of time, t, then the Equation can be adapted by changing Q_s to the number of packets that can arrive in t, which is the packet processing rate, PP_r, multiplied by t. Given the PP_r and using an EWMA the Equation 13 gives the maximum value for measure to guarantee overload will be detected in time t.

$$measure = t * PP_r * (1 - min) * w \tag{13}$$

For these experiments, detecting overload in less than a second is considered reactive, so let $t = 1$. In later experiments, $min = .7$, and $w = .1$. Later experiments also determine that $PP_r \geq 4000$ (note a higher PP_r would give a lower frequency). These values give $measure = 120$.

Given this evaluation and the previous experiment on overhead associated with different measure values, $measure = 125$ is used for the following experiments because it has minimal overhead and is reactive.

6.4 Main Experiments

A set of three experiments was run on varying configurations. The set consists of gradual overload, instant overload, and constant overload.

6.4.1 Gradual Overload

The gradual overload experiment is designed to test how the overload controls react to a steady build up of calls. Initially, two receivers and two senders register with the server. Then, 10 seconds later, the senders start increasing their sending rate from 0 at a constant increase of 100 calls per second. After 50 seconds, the senders hold their sending rate constant at 5000 calls per second (10,000 combined) for the next 30 seconds. Then they

decrease their sending rate at a constant decrease of 100 calls per second. Statistics are kept for the first 120 seconds.

6.4.2 Instant Overload

The instant overload experiment is designed to test how the overload controls react to an instant onset of overload. The two senders and two receivers each initially register with the server. After ten second the senders increase their call rate from 0 to 5000 calls per second almost instantaneously. Statistics are kept for the first 30 seconds.

6.4.3 Constant Overload

The constant overload experiment is designed to test how the overload controls react to constant overload. Initially both pairs of senders and receivers register with the server. Then, after 10 seconds, each sender sets their sending rate to 3000 calls per second. Statistics are kept for the first 70 seconds.

6.5 Base Configurations

Unless otherwise specified, all the experiments are running using the base configuration for each of the overload controls. The parameters ρ_{targ}, f_{min}, and α_{targ} are based on those in [7]. The parameter *measure* is set so it has low overhead while still being fairly reactive. The parameter k is set to get reasonable values for processor occupancy. The parameter w is chosen based on keeping the algorithm reactive without letting transient bursts affect it. The value for w in processor occupancy is the same as in [6].

Different parameters were experimented with and it became apparent that the overload controls could for the most part have the same effect given the proper tuning. These different values were used because they were either used in previous work or are intuitive. These parameters also allow the exploration of different levels of strictness with overload controls. Occupancy is the least aggressive of the overload controls, followed by SiRED which is moderately aggressive, and ARO which is very aggressive. It should also be noted SiRED was the easiest of the overload controls to tune.

No Server The no server configuration did not use the server and had the senders and receivers directly communicate. This is useful for determining the effect of the server in general and for demonstrating that the system is actually overloaded.

Basic The basic configuration runs the server with no overload controls.

Occupancy The occupancy configuration sets the parameters for the occupancy algorithm as follows.

k	25
$measure$	125
ρ_{targ}	.95
f_{min}	.05
w	.1

SiRED The SiRED configuration sets the parameters for the SiRED algorithm as follows.

$measure$	125
f_{min}	.05
Q_{min}	.7
Q_{max}	.9
w	.05

ARO The ARO configuration sets the parameters for the ARO algorithm as follows.

k	25
$measure$	125
ρ_{targ}	.95
α_{targ}	.95
f_{min}	.05
w	.1

6.6 Single Class

6.6.1 Constant Overload

The single class constant overload experiment was run 25 times; the averages of these runs are used. Figure 7 gives the calls per second (CPS) summed from both receivers over the course of the runs. Figure 8 gives the latency in milliseconds over the course of the runs. Figure 9 gives the average latency.

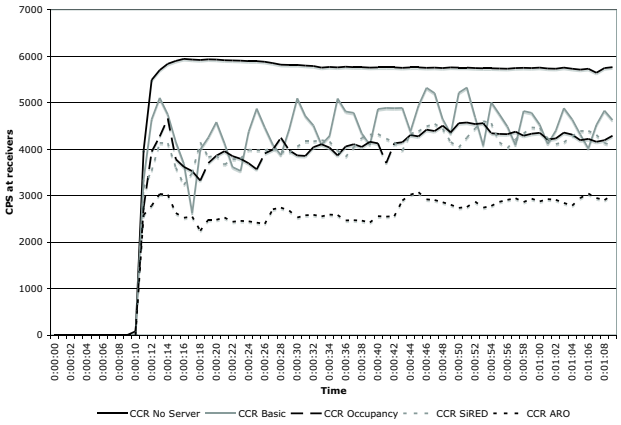

Figure 7: **Single Class Constant CPS**

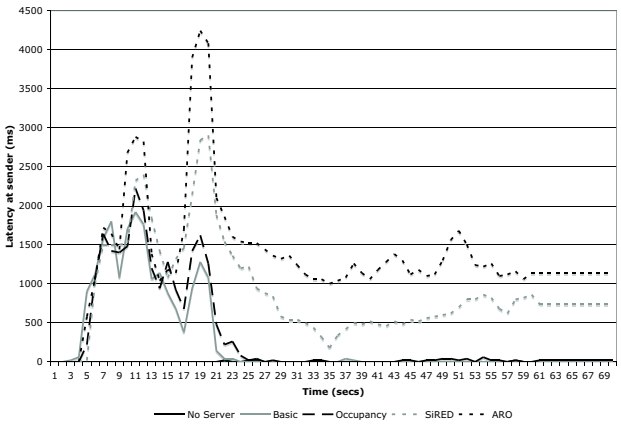

Figure 8: **Single Class Constant Latency**

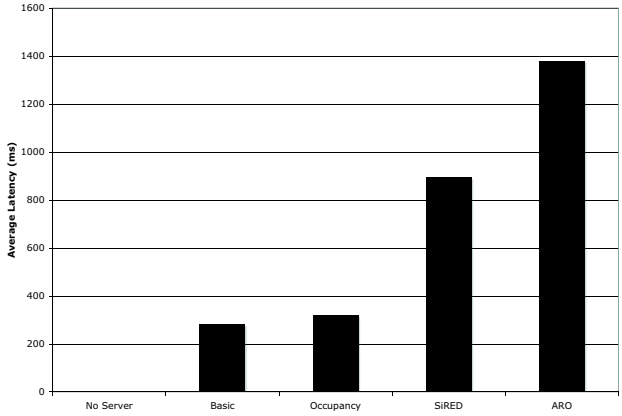

Figure 9: **Single Class Constant Average Latency**

The CPS curve for the server-less configuration has a throughput of approximately 1000 more CPS than the next highest, which is the basic configuration. This confirms the server is actually being overloaded. Basic has the highest throughput followed by occupancy and SiRED, which are similar, and ARO, which is significantly worse. As mentioned previously, occupancy is the least aggressive and ARO the most aggressive. This is the cause of ARO's much lower throughput. SiRED, however, is very similar to occupancy in call throughput, suggesting overload controls can be moderately aggressive without a significant loss in CPS.

The CPS for all the overload controls is smoother than that of the basic configuration. Becasue the lower limit of basic is essentially the occupancy and SiRED curves this has no benefit. In the worse instance for basic, it has essentially the same CPS as the overload controls, while in the best case, it has higher throughput. The smoother behavior of the overload controls is desirable if the average CPS was similar; however this is not the case for this experiment.

The constant experiment is meant to examine how the different configurations performed under steady load. Once the overload controls adjust to the overload, basic and occupancy have minimal latency, SiRED has approximately 500 millisecond latency and ARO has approximately 1250 millisecond latency. The result indicates the more aggressive the overload algorithm, the higher the latency. Intuitively this makes sense because more aggressive overload controls drop more packets, which causes more retransmissions. The retransmissions then significantly increase the latency.

6.6.2 Gradual Overload

The single class gradual overload experiment was run 25 times; the averages of these runs are used.

Figure 10 gives the calls per second (CPS) summed from both receivers over the course of the runs. Figure 11 gives the latency in milliseconds over the course of the runs. Figure 12 gives the average latency.

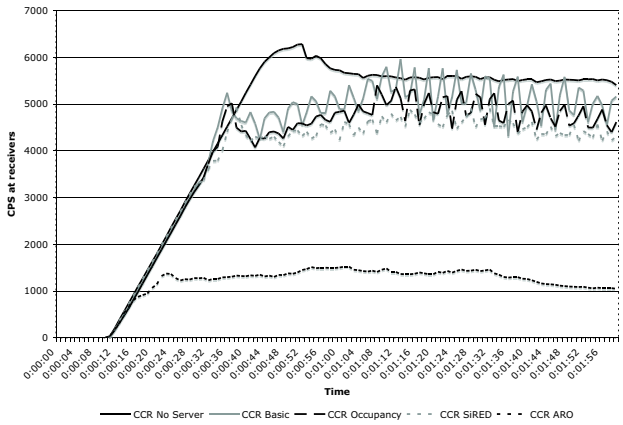

Figure 10: **Single Class Gradual CPS**

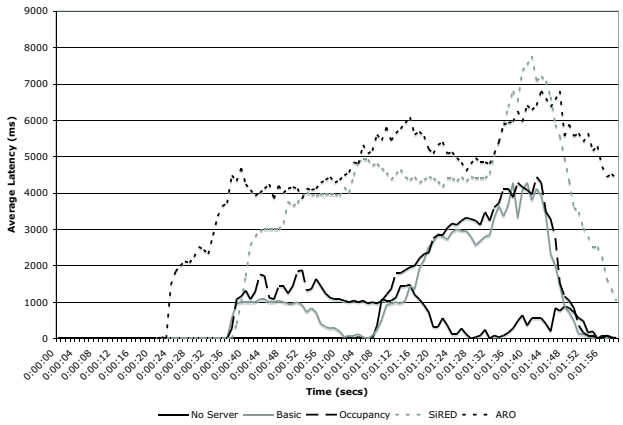

Figure 11: **Single Class Gradual Latency**

The basic curve spikes above the no-server curve a few times; however, becasue the average is not as high the server is overloaded.

ARO is noticeably worse than all the other configurations. Its throughput is lower to a greater degree than in the constant experiment. This indicates very aggressive overload controls perform especially poorly under gradually increasing load. Similar to the constant experiment, the basic configuration has the best CPS followed closely by occupancy and

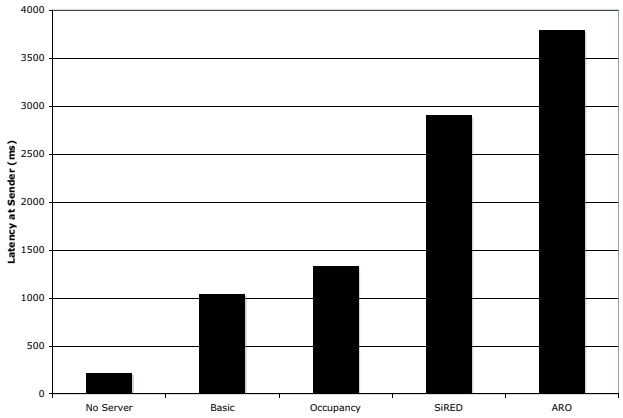

Figure 12: **Single Class Gradual Average Latency**

SiRED.

The latency graph again shows the correlation between aggressiveness and latency. The relationship between CPS and latency can be observed near the twenty four second mark where ARO starts to have much lower CPS and much higher latency. This suggests that, for single class overload control algorithms, with retransmissions, the more aggressive an algorithm the lower the CPS and the higher the latency.

6.6.3 Instant Overload

The single class instant overload experiment was run 25 times; the averages of these runs are used.

Figure 13 gives the calls per second (CPS) summed from both receivers over the course of the runs. Figure 14 gives the latency in milliseconds over the course of the runs. Figure 15 gives the average latency.

The greater CPS for the server-less configuration demonstrates that the server is actually under overload. All the overload controls almost directly follow the basic curve in this experiment, only scaled down based on their aggressiveness with occupancy being the best and ARO the worst. The curves also indicate that smoothness is directly related to the aggressiveness of the algorithm.

The latency is actually lower on average for occupancy and SiRED than basic. This suggests that in the face of instant bursts of load, overload controls can slightly reduce latency on average.

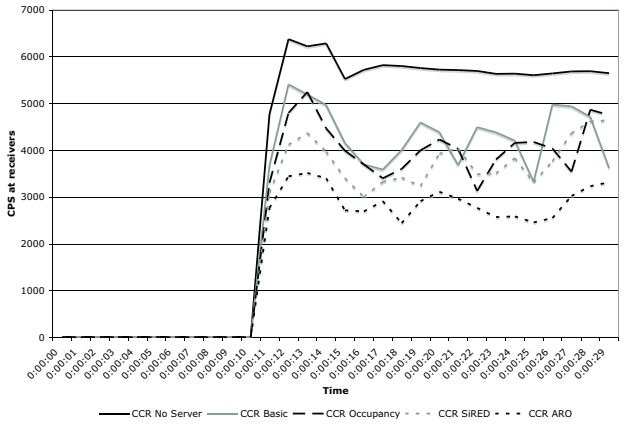

Figure 13: **Single Class Instant CPS**

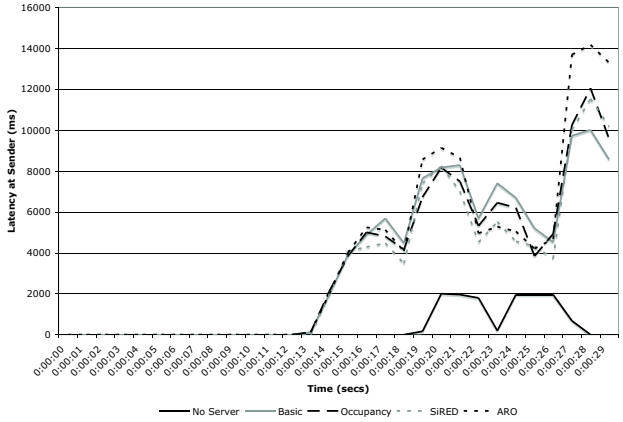

Figure 14: **Single Class Instant Latency**

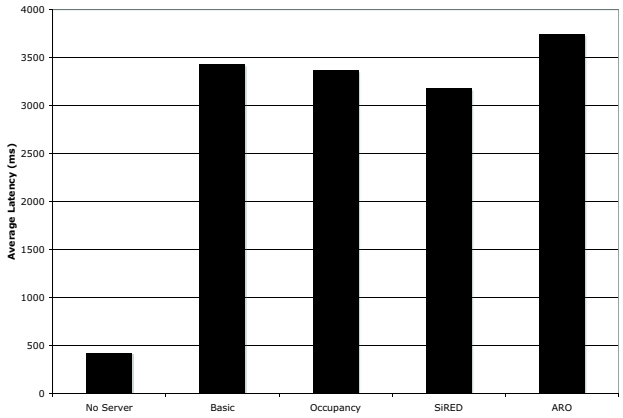

Figure 15: **Single Class Instant Average Latency**

6.6.4 Conclusion

Single class overload controls with retransmissions have little to no benefit. The only experiment where overload controls had any benefit was instant overload where the occupancy and SiRED overload controls had minimally lower call latency. For all other experiments the basic configuration, which has no overload controls, had the highest CPS and lowest latency. Thus, without the ability to suppress retransmissions, single class overload controls should not be used.

6.7 Single Class without Retransmissions

The overload controls suffered high latency under the single class experiment because of retransmissions. It is possible to stop retransmission by either modifying the endpoints (e.g. phones), or by sending a 503 Service Unavailable message. To investigate the benefits that overload controls pose when retransmissions can be eliminated, the previous set of experiments was repeated without any retransmissions.

6.7.1 Constant Overload

The single class constant overload without retransmissions experiment was run five times; the averages of these runs are used.

Figure 16 gives the calls per second (CPS) summed from both receivers over the course of the runs. Figure 17 gives the latency in milliseconds over the course of the runs. Figure 18 gives the average latency.

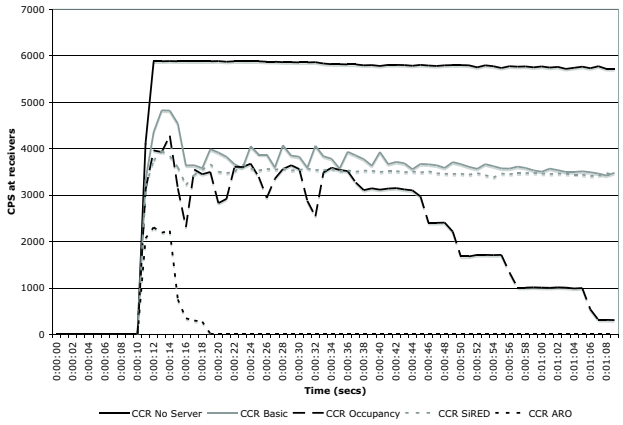

Figure 16: **Single Class without Retransmissions Constant CPS**

The CPS for the basic configuration is the best, followed closely by SiRED. Occupancy is

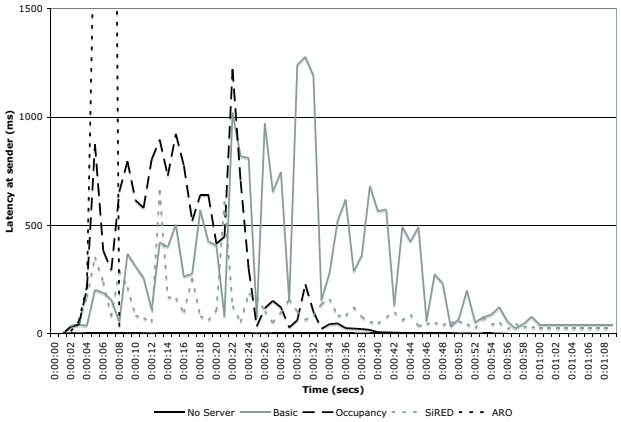

Figure 17: **Single Class without Retransmissions Constant Latency**

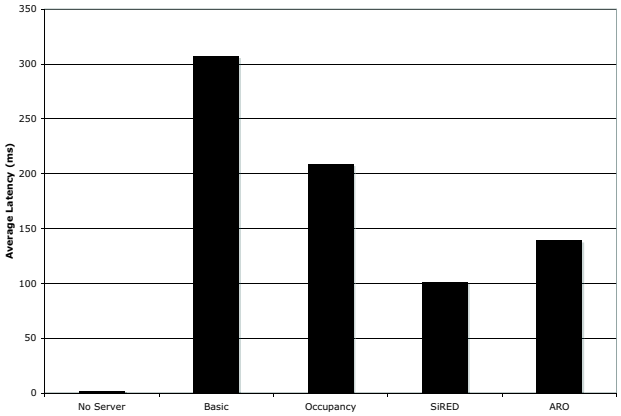

Figure 18: **Single Class without Retransmissions Constant Average Latency**

slightly worse than SiRED for the first 40 seconds, then becomes significantly worse. ARO is terrible and quickly drops to zero CPS. The reason for occupancy's and ARO's CPS dropping to zero is two fold. First, SIPp limits the number of open calls to three times the call rate. An open call is a transaction for which an INVITE has been sent and call tear down has not occurred. So for this scenario, SIPp could only have 18,000 open calls. Once ARO and occupancy hit the limit their CPS dropped to zero. With actual users as opposed to SIPp generating the calls, this would not necessarily happen.

The other reason ARO and occupancy have much lower CPS is that they are more aggressive. For a call to be successfully setup, an INVITE and a 200 OK message must pass through the server. For the same call to be successfully torn down, an ACK and a 200 OK must also through the server. Because there are no retransmissions, each message only has one chance to not be dropped. So given f, the fraction allowed, the probability a call will be successfully set up is f^2. When $f = f_{min} = .05$, the probability of a successful call set up is .0025, meaning on average only 1 out of every 400 calls will be successfully set up. This dramatically decreases the CPS. The probability of a call being successfully set up and torn down (completed) is f^4. When $f = f_{min} = .05$, the probability of a completed call is $6.25 * 10^{-6}$, meaning on average only 6.25 calls out a million will be completed. With such a small chance of completing a call, the 18,000 call limit is quickly reached for both the occupancy and ARO configurations.

Latency for ARO and occupancy drops to zero when the CPS drops to zero. Because there are no calls, there can not be any call latency. Basic has significantly higher latency than SiRED on average and has more variance.

SiRED has very similar CPS to basic, while having significantly lower latency. This indicates under constant overload, moderately aggressive overload controls have a positive effect.

6.7.2 Gradual Overload

The single class gradual overload controls without retransmissions experiment was run five times; the averages of these runs are used.

Figure 19 gives the calls per second (CPS) summed from both receivers over the course of the runs. Figure 20 gives the latency in milliseconds over the course of the runs. Figure 21 gives the average latency. Note that the average latency is computed for only the first 90 seconds, before the open limit is reached.

This experiment confirms a number of previous observations. Like the gradual experiment

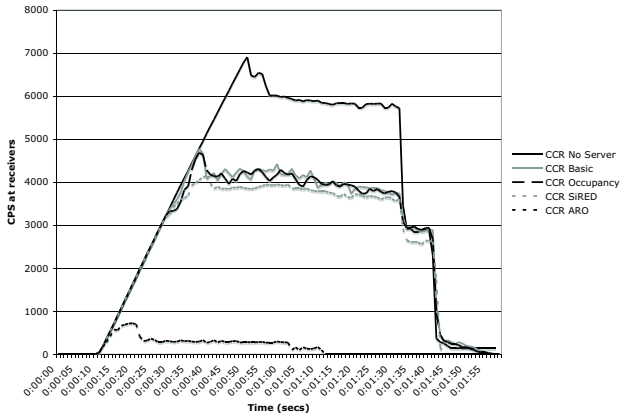

Figure 19: **Single Class without Retransmissions Gradual CPS**

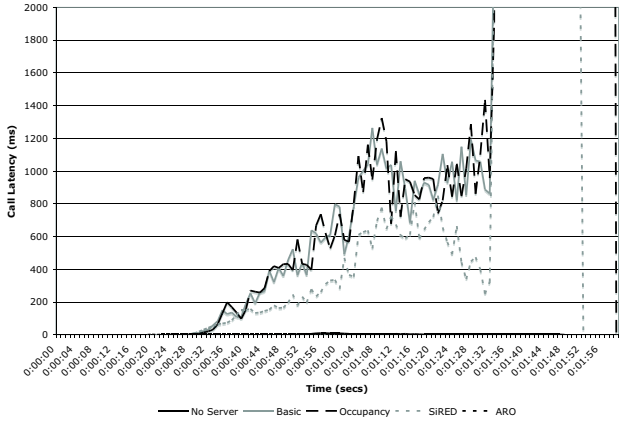

Figure 20: **Single Class without Retransmissions Gradual Latency**

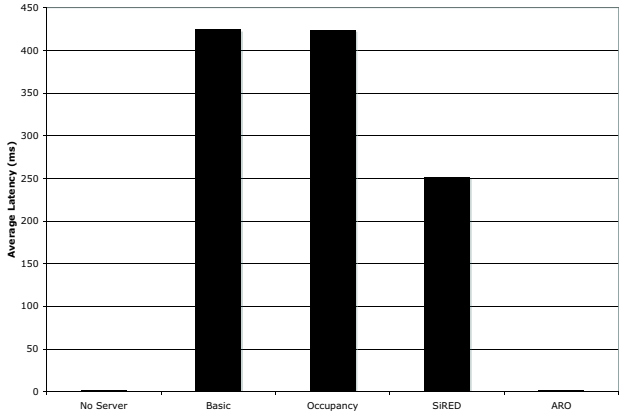

Figure 21: **Single Class without Retransmissions Gradual Average Latency**

with retransmissions, ARO is particularly terrible, confirming that overly aggressive overload controls perform poorly with gradual overload. The large drop in CPS for all configurations including the configuration with no server is caused by the 18,000 open call limit in SIPp. SiRED is again similar to the basic configuration in terms of CPS.

The latency for SiRED is significantly lower than for the basic configuration. This indicates under gradual overload, moderately aggressive overload controls, in particular SiRED, can significantly reduce latency at minimal cost to CPS.

6.7.3 Instant Overload

The single class with retransmissions instant overload experiment was run five times; the averages of these runs are used.

Figure 22 gives the calls per second (CPS) summed from both receivers over the course of the runs. Figure 23 gives the latency in milliseconds over the course of the runs. Figure 24 gives the average latency.

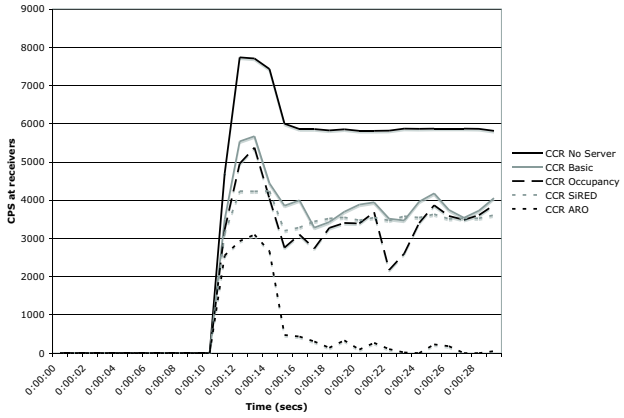

Figure 22: **Single Class without Retransmissions Instant CPS**

The results for the instant experiment confirm the earlier results. SiRED, the moderately aggressive overload control, has similar CPS to the basic configuration, while having lower latency. ARO, the very aggressive overload control, had the worst performance on all metrics, while occupancy, the least aggressive overload, had similar but worse performance than the basic configuration on both metrics.

6.7.4 Conclusion

For single class overload controls without retransmissions, moderately aggressive overload controls (in particular SiRED) can have a positive effect. SiRED had very similar CPS to

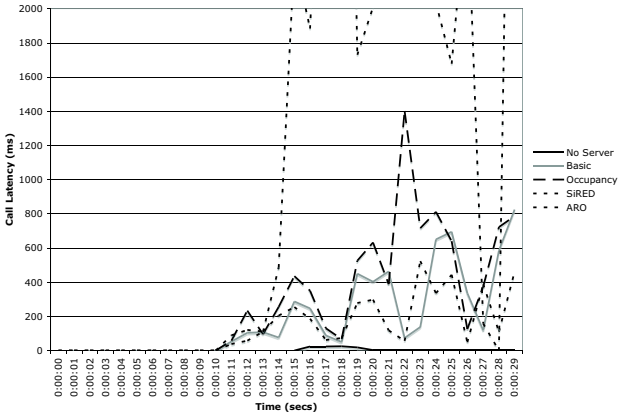

Figure 23: **Single Class without Retransmissions Instant Latency**

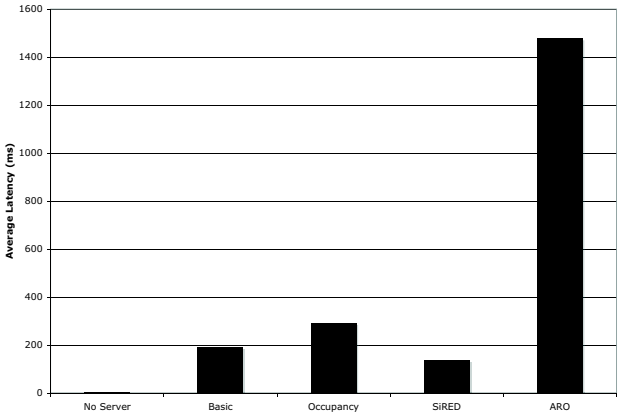

Figure 24: **Single Class without Retransmissions Instant Average Latency**

the basic configuration in all three experiments while having significantly lower latency.

The ability to turn off retransmissions makes single class overload controls a good solution for reducing latency without significantly degrading CPS. The SIP 503 Service Unavailable message turns off retransmissions making an implementation without retransmissions feasible. Sending the message would introduce overhead, which would lower CPS and increase latency, so the benefits should be reevaluated with the overhead of sending 503 messages taken into account. The implementation and testing of a configuration that uses the 503 messages is part of future work.

6.8 Simplified Multi Class

Accepting INVITE messages has significantly higher cost than all other messages. An INVITE will trigger a 180 Ringing, 200 OK, ACK, BYE, and another 200 OK message. By dropping an INVITE, it prevents the future messages from ever appearing. This motivates simplified multi class overload controls, which have only two classes, INVITES and all other messages. The previous experiments are redone with the simplified multi class overload controls.

6.8.1 Constant Overload

The single class constant overload experiment was run 25 times; the averages of these runs are used.

Figure 25 gives the calls per second (CPS) summed from both receivers over the course of the runs. Figure 26 gives the latency in milliseconds over the course of the runs. Figure 27 gives the average latency.

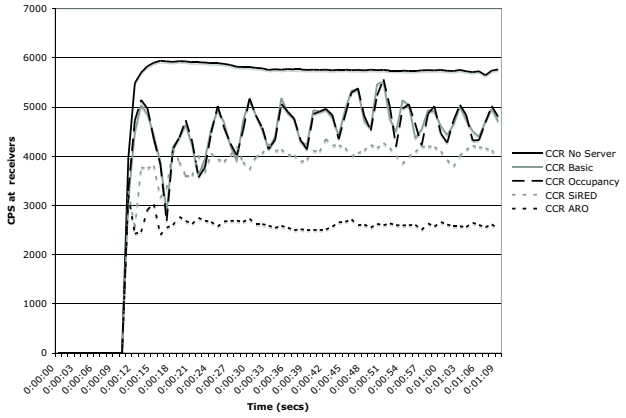

Figure 25: **Simplified Multi Class Constant CPS**

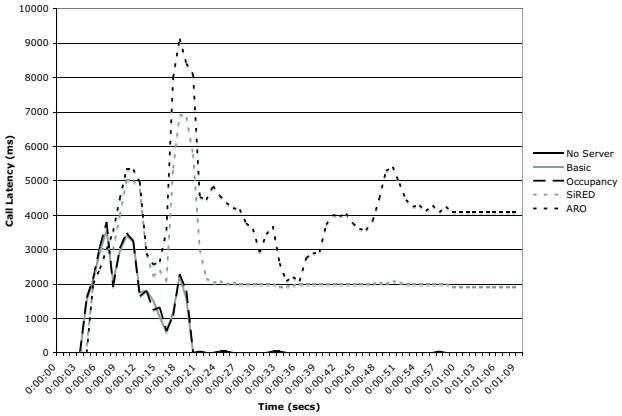

Figure 26: **Simplified Multi Class Constant Latency**

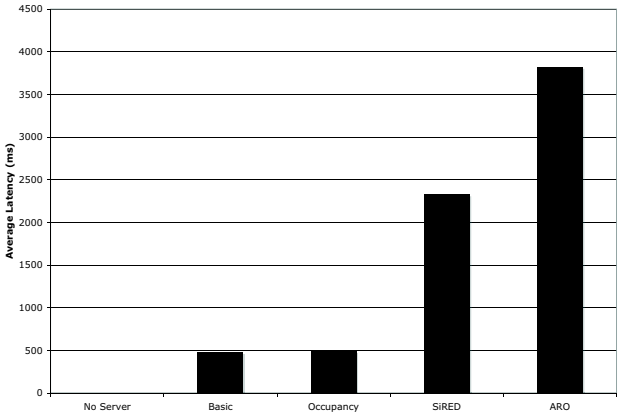

Figure 27: **Simplified Multi Class Constant Average Latency**

The CPS curve for no server is higher than the CPS curve for all other configurations, demonstrating the server is overloaded. The occupancy curve is effectively identical to the basic configuration. This indicates that with measurements every 125 packets, there is no appreciable overhead. SiRED has slightly lower CPS than basic and occupancy, but more stable. ARO, the most aggressive algorithm has very poor CPS again.

The latency curves are similar to the CPS curves. Occupancy is effectively identical to the basic configuration. SiRED has higher latency than the basic configuration, and ARO has even higher latency.

Because the parameters for the overload controls are not changed from the single class case, it is not surprising that occupancy is effectively identical to basic. Occupancy is the least aggressive overload control and this is compounded by only dropping INVITES. The fraction allowed, f, only applies to INVITES, so it should be multiplied by the percentage of incoming packets that are invites for the parameters in simplified multi class to be equivalent to those used in single class.

6.8.2 Gradual Overload

The single class gradual overload experiment was run 25 times; the averages of these runs are used.

Figure 28 gives the calls per second (CPS) summed from both receivers over the course of the runs. Figure 29 gives the latency in milliseconds over the course of the runs. Figure 30 gives the average latency.

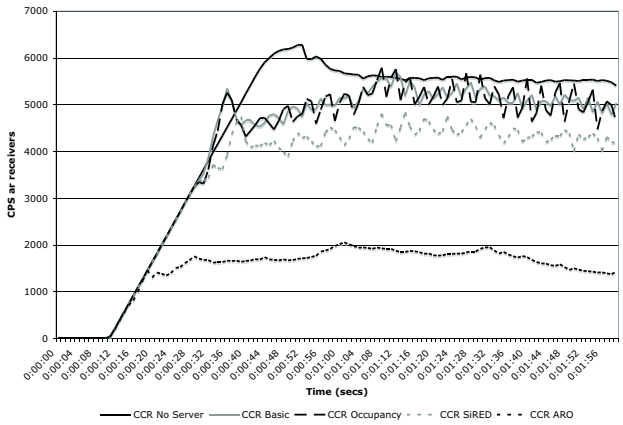

Figure 28: **Simplified Multi Class Gradual CPS**

Occupancy is very similar to the basic configuration for CPS, while SiRED is 15 to 20% worse. ARO has very low CPS as is usual with gradual overload.

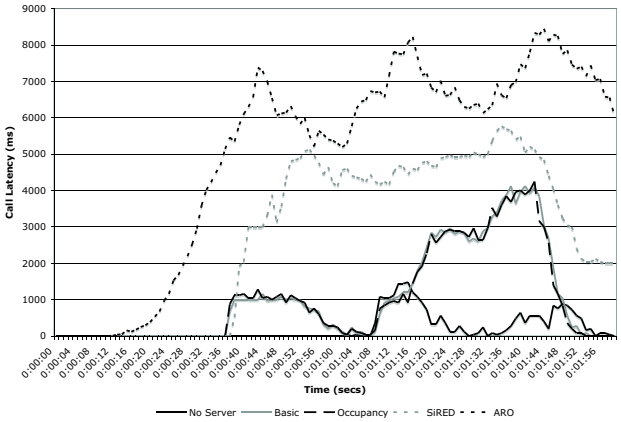

Figure 29: **Simplified Multi Class Gradual Latency**

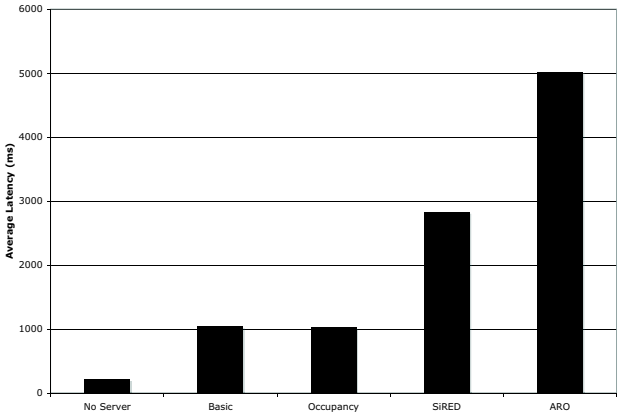

Figure 30: **Simplified Multi Class Gradual Average Latency**

The latency results are similar to the CPS results. Because there are retransmissions, the more aggressive algorithms have higher latency. Occupancy, however, mirrors the basic configuration for latency.

Under gradual overload, the best the simplified multi class overload controls can do is to match the basic configuration.

6.8.3 Instant Overload

The simplified multi class instant overload experiment was run 25 times; the averages of these runs are used.

Figure 31 gives the calls per second (CPS) summed from both receivers over the course of the runs. Figure 32 gives the latency in milliseconds over the course of the runs. Figure 33 gives the average latency.

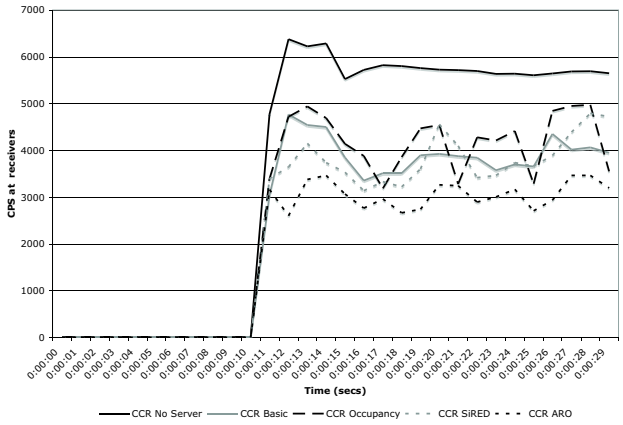

Figure 31: **Simplified Multi Class Instant CPS**

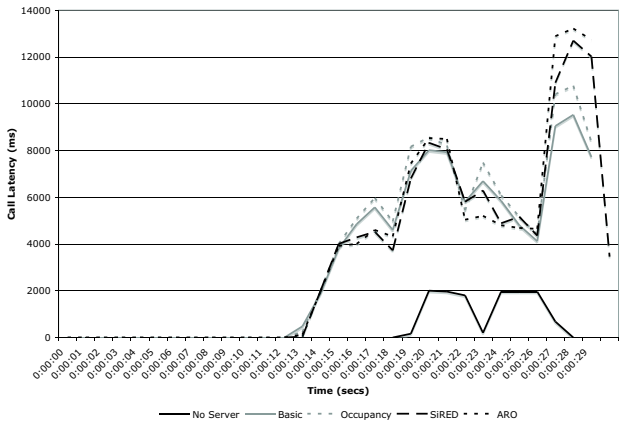

Figure 32: **Simplified Multi Class Instant Latency**

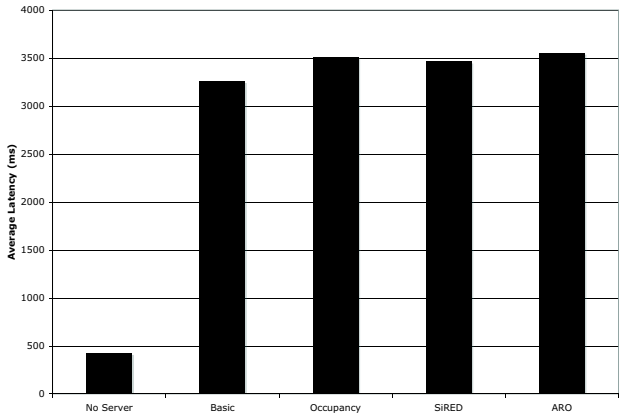

Figure 33: **Simplified Multi Class Instant Average Latency**

The instant overload experimental results are the first indication that overload controls can increase CPS in an actual implementation. Occupancy has higher CPS than the basic configuration, SiRED has very similar CPS to the basic configuration, and ARO has reasonably close CPS. Occupancy's CPS is higher than the basic configuration because a higher percentage of calls that have INVITEs successfully processed have those calls successfully set up.

The latency is minimally higher for the overload controls as compared to the basic configuration. Occupancy has higher CPS at the cost of a minimal increase in latency in the face of instant overload.

6.8.4 Conclusion

Occupancy, the least aggressive of the simplified multi class overload controls can increase the CPS during instant onset of overload with a minimal increase in latency. In both the gradual and constant overload scenarios, occupancy had almost identical CPS and latency to the basic configuration.

Simplified multi class overload controls have a tangible benefit. Specifically, less aggressive overload controls can increase the CPS when instant overload occurs without affecting performance under constant or gradual load.

6.9 Simplified Multi Class without Retransmissions

6.9.1 Constant Overload

The simplified multi class constant overload controls without retransmissions experiment was run five times; the averages of these runs are used.

Figure 34 gives the calls per second (CPS) summed from both receivers over the course of the runs. Figure 35 gives the latency in milliseconds over the course of the runs. Figure 36 gives the average latency.

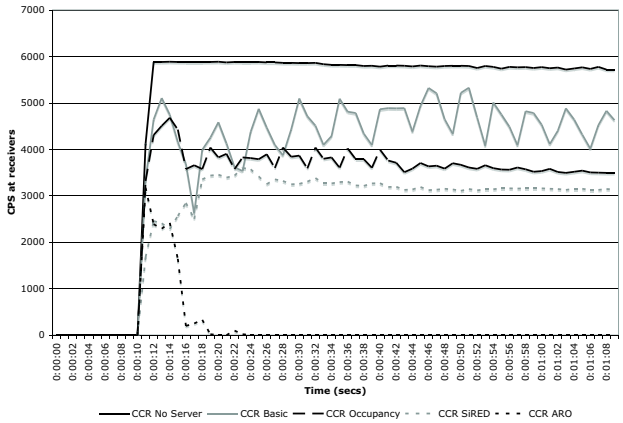

Figure 34: **Simplified Multi Class without Retransmissions Constant CPS**

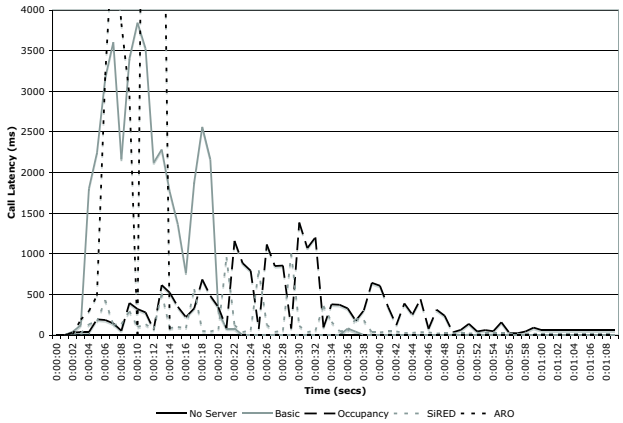

Figure 35: **Simplified Multi Class without Retransmissions Constant Latency**

Under constant overload without retransmissions all the simplified multi class overload controls have noticeably lower CPS than the basic configuration. Occupancy is the closest to the basic configuration, with SiRED having slightly lower CPS, and ARO having very low CPS. ARO suffered from the SIPp maximum open calls problem again.

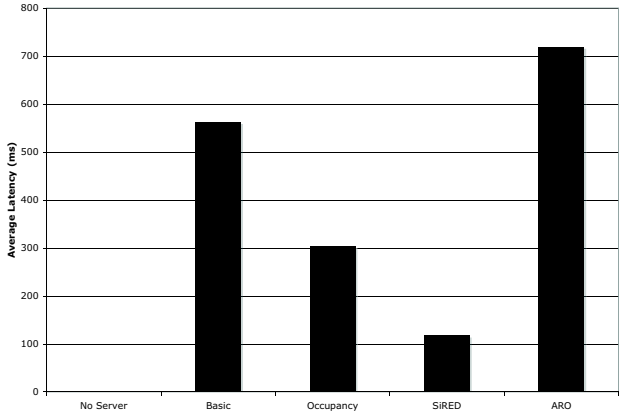

Figure 36: **Simplified Multi Class without Retransmissions Constant Average Latency**

Latency-wise, both occupancy and SiRED significantly outperform the basic configuration. The maximum latency for occupancy is less than half of the maximum for basic, and the maximum latency for SiRED is less than a third. Average latency is significantly less for occupancy and SiRED.

For constant overload, there is a clear trade off between CPS and latency. With an approximately 15% reduction in CPS, average latency can be halved by using the occupancy algorithm. With an approximately 25% reduction in CPS, average latency can be reduced to a fifth by using the SiRED algorithm.

6.9.2 Gradual Overload

The simplified multi class gradual overload controls without retransmissions experiment was run five times; the averages of these runs are used.

Figure 37 gives the calls per second (CPS) summed from both receivers over the course of the runs. Figure 38 gives the latency in milliseconds over the course of the runs. Figure 39 gives the average latency.

Note the first 90 seconds are used for these results. After that time, the SIPp maximum open calls problem comes into play. The gradual results are very similar to the constant results. The basic configuration has higher CPS than occupancy and SiRED, while ARO has very low CPS. Occupancy and SiRED are also noticeably more stable.

Occupancy and SiRED have significantly lower latency than the basic configuration. ARO has almost no latency because it allows so few calls through. The trade off between CPS and latency is clearly shown here.

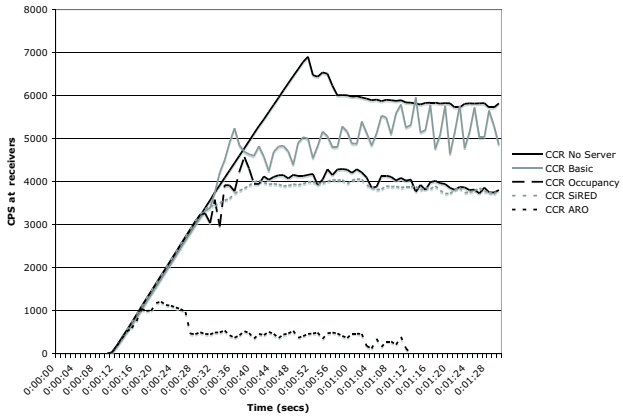

Figure 37: **Simplified Multi Class without Retransmissions Gradual CPS**

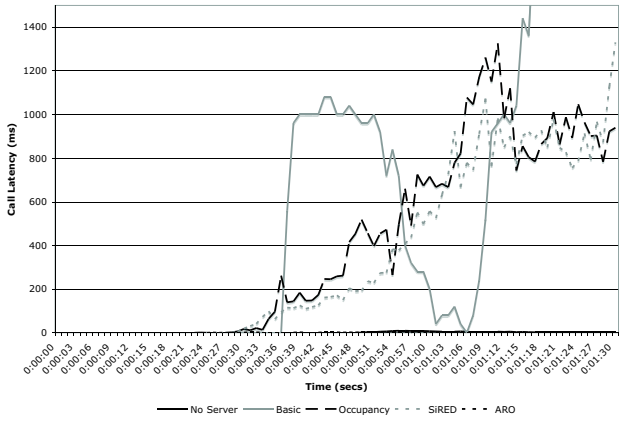

Figure 38: **Simplified Multi Class without Retransmissions Gradual Latency**

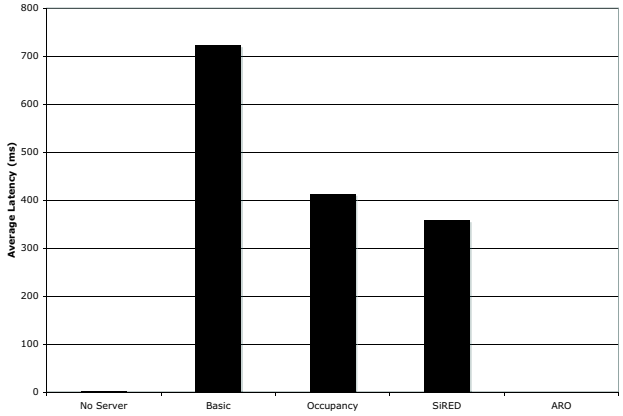

Figure 39: **Simplified Multi Class without Retransmissions Gradual Average Latency**

6.9.3 Instant Overload

The simplified multi class instant overload controls without retransmissions experiment was run five times; the averages of these runs are used.

Figure 40 gives the calls per second (CPS) summed from both receivers over the course of the runs. Figure 41 gives the latency in milliseconds over the course of the runs. Figure 42 gives the average latency.

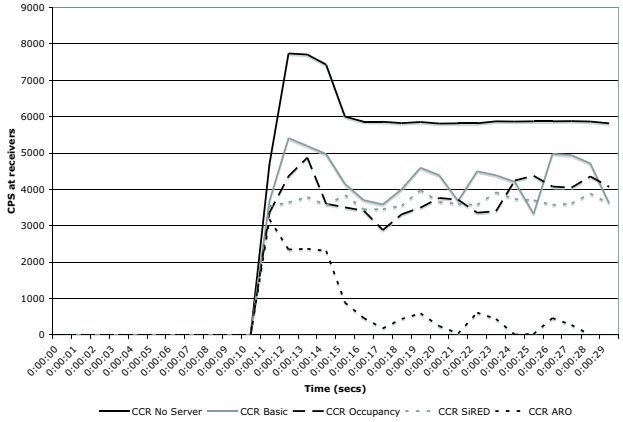

Figure 40: **Simplified Multi Class without Retransmissions Instant CPS**

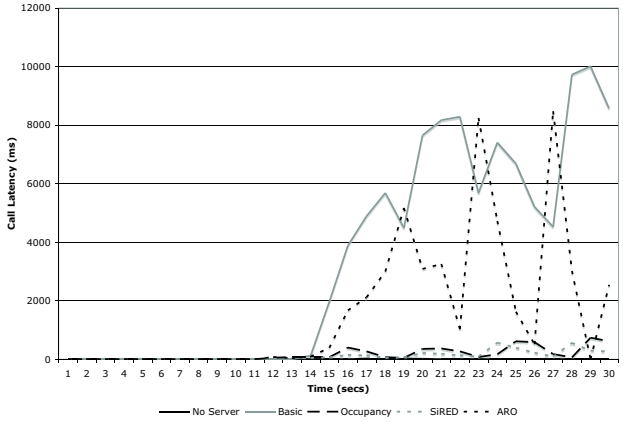

Figure 41: **Simplified Multi Class without Retransmissions Instant Latency**

Under instant overload the simplified multi class overload controls perform their best. The CPS for occupancy and SiRED is very similar to that of the basic configuration.

The latency for calls in occupancy and SiRED is drastically better than that for the basic configuration. For both occupancy and SiRED the average latency is less than a tenth of the basic configuration. Thus under instant overload, simplified multi class controls can dramatically reduce latency with only a small decrease in CPS.

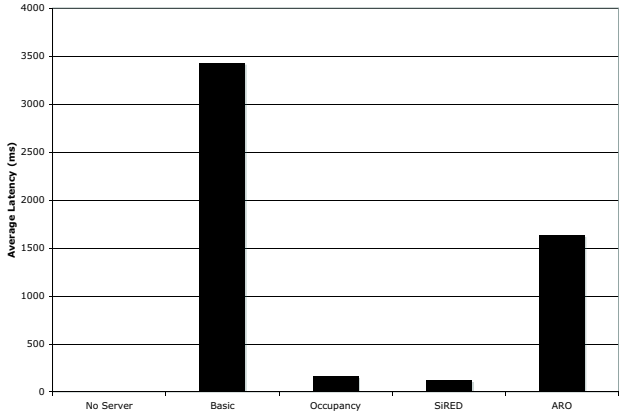

Figure 42: **Simplified Multi Class without Retransmissions Instant Average Latency**

6.9.4 Conclusion

For all experiments, simplified multi class overload controls without retransmissions can significantly reduce latency. The cost in terms of CPS varies based on the different experiment. Under instant load it is very low, while under gradually increasing load it is 15% to 25%. The decrease in latency is significant under all experiments, but it is incredible under instant load.

6.10 Analysis

Code instrumentation indicates that overload controls add approximately .2 microseconds of overhead per packet and approximately 1 microsecond per packet when they make a measurement. Fully processing a packet takes 9.68 to 26.69 microseconds. Dropping a packet takes 2.75 microseconds when done at the application level and 1.1 microseconds in the kernel. This shows that dropping a packet saves a large amount of processing time at the application level and an even greater amount in the kernel.

Experimental results indicate that when measurement are done less frequently than once every 125 packets, the overhead is minimal. Also, when the measurement is done more frequently than once every 125 packets, the overload controls are reactive. Thus, measuring once every 125 packets allows reactive overload controls with negligible overhead.

Single class overload controls have no benefit with retransmissions; however, with the ability to suppress retransmissions overload controls can significantly reduce call latency at a small cost in terms of CPS.

Simplified multi class overload controls can increase CPS with a small increase in latency

with retransmission. Without retransmissions overload controls can dramatically reduce call latency with a small decrease in CPS.

7 Related Work

S. Kasera et al.[6] introduce the idea of reactive overload controls for the traditional telephony network. They state that switches in that network have overload controls designed for sustained high load and do not react well to sudden bursts of load. This motivates their design of three overload controls algorithms. Their three overload control algorithms are the precursors to the algorithms in their later work Kasera et al.[7], which form the basis for the overload controls implemented in this thesis. S. Kasera et al.[6] differs from this thesis significantly. It is targeted at switches in the traditional telephony network, this thesis is targeted at SIP servers in a VoIP network. It considers all packets as single class, this thesis examine both single and simplified multi class algorithms. It uses simulations to demonstrate its results, while this thesis implements the overload control in a server and present that server with actual load.

S. Kasera et al.[7] builds on their previous work and introduces the idea of multi class overload controls for switches in the traditional telephony network. They also define the concept of "equivalent system load measure" that accounts for the differences in call processing time for different classes. The overload control algorithms used in this thesis directly correspond to those present in this work. The differences are that this thesis is targeted at SIP server not traditional telephony switches, and that this thesis implements the overload controls and tests them in a real system as opposed to a simulation. A number of the assumptions in S. Kasera et al.[7] are thus stripped away.

V. Balasubramaniyan et al.[2] observe that stateful processing of SIP messages significantly increases processing overhead. This motivates their design of SERvartuka, which dynamically changes the fraction of calls handled statefully and statelessly. They implement SERvartuka in OpenSER, a derivative of SER, and measure the benefits on a IBM BladeCenter. This work considers only the fraction of calls to handle statefully vs statelessly and does not examine the costs and benefits of dropping calls. Future work could incorporate their work with the work presented in this thesis to create a combined approach. This combined approach would have three tiers, where calls could either be handled statefully, handled statelessly, or dropped.

In Acharya et al.[1] the authors implement an in kernel packet classifier that assigns packets to different queues based on their user defined value. The authors note that the packet classifier is ideally suited for implementing overload controls since it has low overhead and is early in the processing path of incoming packets, so dropping saves more processing

time. Future work involves integrating the overload controls presented in this thesis with the classifier present in Archarya et al.[1].

M. Ohta[8] looks at separating SIP messages into two classes, INVITEs and all other messages and applying overload controls to INVITE messages only. This is an intelligent approach and is also adopted in this thesis. The overload control suggested in the paper is similar to SiRED proposed in [6] and [7] and used in this thesis. No other overload controls are given. This paper evaluates the overload control using a network simulator ns-2, and thus is not as realistic as is this thesis.

R Ezjak et al.[5] compare ISDN User Part signaling, which is used in traditional telephony networks with SIPp. They note that a server may send a 503 message to other nodes with a time value that indicates how long the other nodes must wait until they can send another INVITE message to the server. This does apply in the situation this thesis studies and is the motivation for the experiments without retransmissions. This approach could also be useful for alleviating load being passed onto the server from other servers. In addition, future work will look at the costs and benefits of sending this 503 message.

8 Conclusion

This thesis implemented and investigated the advantages and drawbacks of single and simplified multi class overload controls for a SIP server. SIP a signaling protocol used mainly for VoIP application. A SIP server is used to provide user location, authentication, and many other features.

When packets arrive at a SIP server at a rate greater than the rate packets can be serviced for an extended period of time, the server is *overloaded*. Overload controls aim to detect and mitigate overload by either increasing the CPS or decreasing the call latency.

This thesis implemented overload controls similar to those in [6] and [7]. They were implemented as a module to SER, an open source SIP server. A combination of four computers was used to overload the server, using SIPp, an open source packet generator.

All the overload controls could be tuned by adjusting their parameters to have similar effects. However, occupancy was naturally less restrictive, SiRED moderately restrictive, and ARO very restrictive. In addition, SiRED was the easiest to tune to change restrictiveness.

Results showed that dropping calls at the application level saved a large amount of processing time and that dropping in the kernel would save even more time. Overload controls could have negligent overhead but still be reactive when measurement were taken once every 125 packets.

Single class overload controls could reduce latency with a minimal decrease in CPS when suppressing retransmissions. Simplified multi class overload controls could increase CPS without suppressing retransmissions or dramatically reduce latency when suppressing retransmissions.

9 Acknowledgments

I would like to thank Arup Acharya and Charles P. Wright from IBM Research for our many discussions. I would also like to thank John Hannan for his helpful comments and review. I wish to thank Kristine Snodgrass for copy editing this thesis. Finally, I wish to thank Thomas La Porta, my adviser, who provided a tremendous amount of guidance for this thesis.

References

[1] Arup Acharya, Xiping Wang, and Charles P. Wright. A sip classification engine for revenue-maximizing overload control of sip servers. Research report RC24022, IBM T. J. Watson Research Center, August 2006.

[2] Vijay A. Balasubramaniyan, Arup Acharya, Mustaque Ahamad, and Charles P. Wright. Servartuka: Enhancing sip server scalability with dynamic state management, March 2007.

[3] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Trans. Netw.*, 1(4):397–413, 1993.

[4] Audio-Video Transport Working Group, H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. Rtp: A transport protocol for real-time applications. Technical report, , United States, 1996.

[5] Jeroen van Bemmel Harold Batteram, Erik Meeuwissen. Sip message prioritization and its applications. Technical Report 1, Bell Labs Technical Journal, Bell Labs Europe, Hilversum, The Netherlands, 2006.

[6] Sneha Kasera, Jose Pinheiro, Catherine Loader, Mehmet Karaul, Adiseshu Hari, and Tom LaPorta. Fast and robust signaling overload control. In *ICNP '01: Proceedings of the Ninth International Conference on Network Protocols*, page 323, Washington, DC, USA, 2001. IEEE Computer Society.

[7] Sneha Kasera, Jose Pinheiro, Catherine Loader, Tom LaPorta, Mehmet Karaul, and Adiseshu Hari. Robust multiclass signaling overload control. In *ICNP '05: Proceedings of the 13TH IEEE International Conference on Network Protocols (ICNP'05)*, pages 246–258, Washington, DC, USA, 2005. IEEE Computer Society.

[8] Masataka Ohta. Overload control in a sip signaling network. *Transactions on Engineering, Computing and Technology*, 12:205–209, March 2006.

[9] http://www.intel.com/cd/ids/developer/asmo-na/eng/209859.htm?page=2.

[10] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. Sip: Session initiation protocol. Technical report, , United States, 2002.

[11] http://www.iptel.org/ser/.

[12] http://sipp.sourceforge.net/.