Efficient algorithms for computing resource availability in WAA environments

Shimon Golan
Efficient algorithms for computing resource availability in WAA environments

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science

Shimon Golan

Submitted to the Senate of
the Technion — Israel Institute of Technology
Adar Bet 5765 Haifa March 2005
The research thesis was done under the joint supervision of:

Dr. Avigdor Gal, Faculty of Industrial Engineering and Management

Prof. Ron Y. Pinter, Faculty of Computer Science

I wish to express my deepest thanks to Dr. Avi Gal and Prof. Ron Pinter for their guidance, understanding and support throughout the work on the thesis, and especially at the final stages when I needed it the most.

Thanks to my parents, Shalom and Lea, and my sister, Ronit for their enduring encouragement and support. Special thanks to my mother who believed in me and never ceased to push me forward.

And last but not least, I wish to thank my dear friend Hila Yehuda - your love and care kept me going.

The generous financial help of the Technion is gratefully acknowledged.
Contents

Abstract 1

Abbreviations and Notations 2

1 Introduction 3

2 Background and Related Work 7

3 Model Description 10

4 Comparative Framework for Resource Lifespan Estimation 16
   4.1 The Uniform Validation Scheme 20
   4.2 The EpDist Ep Calculation Scheme 22

5 The NonUniform and EpCombined Algorithms 25
   5.1 The NonUniform Validation Algorithm 25
      5.1.1 Data Structures 26
      5.1.2 Main Loop of NonUniform 29
      5.1.3 Re-scheduling Future Resource Probing 32
      5.1.4 Rank Calculation 47
   5.2 The EpCombined Algorithm 50
5.2.1 Server Reply Statistics Gathering ......................... 51
5.2.2 \( E_p \) Calculation Scheme ............................... 52

6 Simulation .............................................. 58
   6.1 Simulation Set Up ....................................... 58
   6.2 No Probing Results ..................................... 63
   6.3 Rank Calculation Function Empirical Tuning ............. 64
      6.3.1 Incorporating Resource Existence Probability \( E_p \) into the Rank
            Calculation Function ............................... 69
      6.3.2 Incorporating the Resource Popularity into the Rank Calculation
            Function ............................................. 70
   6.4 Analysis of Results ................................... 72
      6.4.1 Impact of Noise to Mean of Death .................. 73
      6.4.2 Impact of Resources’ Lifespan Distribution ........ 74
      6.4.3 Impact of Server Availability ....................... 77
      6.4.4 Impact of Number of TCP Connections ................ 78
      6.4.5 Impact of Number of Resources Per Server ........... 81

7 Conclusions ............................................. 83
List of Figures

3.1 the servers’ monitoring model ................................................ 10
3.2 the mediator’s working cycle .................................................. 11
3.3 serverAvailability pattern example ......................................... 13

4.1 algorithms’ classification graphical illustration ............................... 19
4.2 reduced algorithms’ classification matrix ..................................... 21
4.3 a Uniform validation scheme .................................................. 23
4.4 Initialize - initialization of the Uniform validation scheme ............... 23
4.5 EpDist calculation scheme ..................................................... 24

5.1 data structures of NonUniform ................................................ 28
5.2 main loop of NonUniform ..................................................... 30
5.3 initialization at start of cycle i ............................................... 30
5.4 select server in NonUniform .................................................. 31
5.5 select resource in NonUniform ............................................... 32
5.6 re-scheduling algorithm re-schedule(r, i) .................................... 34
5.7 schedule(r, b, B) .............................................................. 36
5.8 insert(r, i) ........................................................................ 37
5.9 remove(r, i) ....................................................................... 37
5.10 illustration of the bucket choosing in list B .......................... 38
5.11 run example of reschedule: scheduling list of resources [(4, 4), (4, 3),
(4, 2),(4, 1)] to bucket 3. ............................................. 39
5.12 rank2nextCycle ................................................. 49
5.13 epCombined algorithm ............................................. 51
5.14 $p_{\text{norm}}(t)$ for mean=10, std=2.5 .......................... 55
5.15 $P(t)$ for various times, as function of $p_{\text{ERROR}}$ and $p_{\text{DEAD}}$ [in %] .......................... 57

6.1 the mediator’s working cycle ...................................... 59
6.2 resources’ lifespan distributions .................................. 61
6.3 cost of re-schedule .............................................. 62
6.4 no probing results ............................................... 64
6.5 impact of load and constant next-cycle function on target-function ... 66
6.6 impact of load and constant next-cycle function on total operations ... 67
6.7 resources’ lifespan normally distributed .......................... 68
6.8 the Ep to rank linear decreasing function ........................ 70
6.9 rank setting by Ep - linear decreasing, load=0.3 ................. 71
6.10 step $Ep$ to Rank function ....................................... 72
6.11 rank setting by $Ep$ - step function - for various values of $MaxNCF$ ... 73
6.12 popularity to Rank multiplication factor .......................... 74
6.13 rank setting by combination of $Ep$ and Popularity ............... 75
6.14 simulation results comparison - noise to mean of death ........... 76
6.15 simulation results comparison - varying $std/mean$ of resources’ lifespan,

$noise-to-mean-of-death=0.2$ ...................................... 77

6.16 simulation results comparison - varying $std/mean$ of resources’ lifespan,

$noise-to-mean-of-death=-0.5$ ...................................... 78

iv
6.17 simulation results comparison - varying server availability    79
6.18 simulation results comparison - varying number of TCP connections . 80
6.19 simulation results comparison - varying number of resources per server 82
Abstract

Wide Area Applications (WAA) involve hundreds of servers and tens of thousands of clients. In such systems, the need to monitor the availability of resources residing at servers is of great importance for the sake of supplying high quality of service to the clients. To this end, we propose an algorithm for background active collection of resource availability patterns, assuming that resources have a limited lifespan.

First, we present a model which describes a WAA environment and show that without periodic probing of the resources in the WAA, the quality of service delivered to the clients is poor. Then we develop an algorithm that periodically probes resources to study their availability status, and which adapts its probing frequency online according to resource popularity and dynamic availability status changes. The estimates of resource availability take into account both basic resource characteristics and the history of previous probings and therefore is adaptive to changes.

We show by simulation that this algorithm outperforms common alternatives. Specifically, we show that it does better than uniform probing which was found to give good results in the different but close area of web crawling.
Abbreviations and Notations

TCP — a data transmission protocol on networks
FTP — a file transfer protocol
HTTP — the hypertext transfer protocol used to view web pages
LRU — the least recently used selection algorithm
WAA — wide area applications
dt\_r — lifespan of resource r
\(\hat{dt}_r\) — estimated lifespan of resource r
pop\_r — popularity of resource r
r.Ep — existence probability of resource r
r.age — a tuning parameter to prevent starvation
MAXQ — size of the window in NonUniform
RQ — window held per server in NonUniform
SC — global window of Adaptvie
threshold — Ep death threshold
serverAvail — server responsiveness in [0, 1]
ts — estimated time to send a request
tp — estimated time to process a response
serverLat — server’s response latency time
Chapter 1

Introduction

With the immense volumes of data sources continuously created, updated and eventually retired from service on the web, the need to supply better quality of service to the client arises. This need is shared by many prevalent middleware utilities such as the Handle system [20] which is a general-purpose global name service that allows secured name resolution and administration over the Internet. To enhance quality of service, multitude of mediators were proposed, both in academia and in industry. A typical system comprises:

- **Servers** are service providers (such as HTTP servers, FTP servers, etc.). Each server contains a multitude of resources, e.g., a Web page in an HTTP server or a file in an FTP server.

- **Clients** are distributed around the network. Each client has local caches with information on resources, such as a timestamp, indicating the resource’s update time. Clients may be tolerant towards data obsolescence, thus allowing the retrieval of cached data even if it has been updated at the server already [5].

- **Mediators** mediate among clients and servers. Clients do not access servers directly. Rather, they access a mediator which redirects them to the relevant servers or sends
them back to their local caches (if the required information is already cached and still considered to be valuable by the clients). The mediator aims at optimizing query delivery time, which is assumed to have a greater latency compared to that of the client/mediator interaction. The policy according to which mediators redirect client requests is based on clients’ information, including the content of local caches and obsolescence tolerance.

Wide Area Applications (WAA), denoting applications that involve hundreds of servers and tens of thousands of clients, serve as a main motivation for our research. In such systems, the need to monitor availability and staleness of resources residing on the servers is of great importance for the sake of supplying high quality of service to the clients.

Here is a simple scenario in a WAA: assume a client $C$ wishes to view a Web page held by some server $S$. Assume that the client had seen this page a week ago and a copy of the page is stored in its local cache. It is worth noting that since a client can be an organizational proxy, it is possible that the previous access to the Web page was performed by a different user within the same organizational unit. Now, the client contacts the mediator (which we assume is readily accessible, due either to location proximity or to the low bandwidth that such a request requires) and asks to download the relevant page from server $S$. In response, the mediator can do one of the following:

- Redirect the client to the server $S$.
- Redirect the client to an alternative server $S'$, if $S'$ has a sufficiently fresh copy of the page, and the latency of downloading the page from $S'$ is lower than that of $S$.
- Inform the client that the copy held within its local cache is within its obsolescence tolerance level. Alternatively, inform the client that the page cannot be refreshed, since $S$ is down, and propose to use the cached copy instead.
The mediator’s responsibility, as demonstrated in the example above, goes beyond the typical services that contemporary mediators provide. Typically, such services require background information on servers’ availability. The aim of this thesis is to propose efficient algorithms for background collection of such knowledge on servers’ patterns of behavior, so that the answers given by the mediator to the client are as accurate as possible.

In particular this thesis focuses on background information that we call **resource availability patterns**. We assume that each resource has a limited lifespan, after which the resource dies and becomes unavailable. A resource can also have periodic down times, for example when the server it resides on malfunctions. We wish to develop a model and an algorithm that estimate the lifespan of each resource based on past probing as accurately as possible. To illustrate the need, note that an undesirable situation occurs if some client requesting a resource is directed to a server which eventually returns a **RESOURCE NOT FOUND** typical error message; this condition is termed **false positive** and abbreviated as **FP**. An equally undesirable situation might occur when the client is told the resource is unavailable while it actually is; this is called **false negative** and abbreviated as **FN**.

The trivial solution of probing all servers all the time does not scale. Also, simple round-robin methods are likely to fail due to the different lifespans of resources. Therefore, our solution is based on a combination of a stochastic estimation of a resource lifespan, combined with feedback from earlier probing tasks. We partition the monitoring task into two subtasks, namely deciding on the availability of a resource, and determining the next time to probe. The first subtask may be hampered by indecisive response to probing the resource. In the proposed model, we differentiate among three possible responses to a probing task, namely **OK**, **DEAD**, and **ERROR**. **OK** means that the resource is available, and **DEAD** means that the resource became permanently unavailable. **ERROR** is
an indecisive response, e.g., due to server maintenance. In case of an ERROR response, we utilize a stochastic model in determining availability. The second subtask is constrained by the availability of resources for background monitoring. As such, we optimize the probing schedule by defining a measure of importance for each resource (termed rank) and heuristically translate the rank of a resource into a probing schedule.

To support the benefit of our approach, we have designed and implemented a simulation environment, in which a mediator monitors resources at servers and estimates their availability. Our empirical analysis shows that the combined use of an a-priori stochastic modeling of resource lifespan, combined with rank-based monitoring yields low error measure, when contrasted with models that are solely based on stochastic analysis of lifespans, or that monitor in a round-robin fashion, ignoring the expected lifespan of resources altogether.

The rest of the thesis is organized as follows. Chapter 2 discusses related work: though no previous work tackled this specific task of resource availability on the Web, the area of Web crawling is quite related and we shall describe work done there. Chapter 3 describes our model. The following two chapters discuss the algorithms and their analyses, where Chapter 4 gives a general categorization of the algorithms and Chapter 5 focuses on the Uniform and NonUniform algorithms. Heuristics related to certain elements of the algorithms are further developed in Chapter 6, which also provides a thorough empirical analysis based on simulation results. We present final conclusions in Chapter 7.
Chapter 2

Background and Related Work

Related research can be classified into three main contexts. First, continuous query systems ([14], [15], [16]) support long-term querying, assuming data is pushed in the form of data streams [3]. Web monitoring, however, primarily uses polling techniques [17].

Second, Web crawlers incrementally update local indices by periodically surveying the Web. Cho and Garcia-Molina [8, 9] performed a large study of the Web, spanning a period of a few months and covering over half a million Web pages. According to their analysis, the lifespan of a Web page can be as brief as a few days (10% overall and maximum of 15% in the .com domain), and as long as over 4 months (30% overall and maximum of 50% for the .gov domain). These results serve as a motivation for our work. Their findings also indicate that a uniform allocation policy, which visits pages at uniform frequency, disregarding the actual change frequency, performs better than a proportional allocation policy for tracing updates to Web pages. This finding may be context dependent, though. For example, in [12], the uniform policy was shown to be inferior to proportional probing in the context of mail servers and postings to newgroups. Our work also shows that for minimizing misestimation of lifespans, the uniform method is inferior to proportional
probing.

In the same area of Web crawling, Edwards et al. [10] proposes a crawler which minimizes the number of obsolete pages without any a-priori assumptions regarding the change frequency of Web pages, relying rather on historical information collected from the pages and maintained as meta data. Such information gathering may be hampered by indecisive responses, similar to the ERROR message discussed in this work. They pose an optimization problem which is highly nonlinear and used the NEOS public server [1] for the solution.

The third area of research discusses Web monitoring for updates, and is closest in spirit to our approach. Pandey et al. [17], extending the work in [18], introduce a general-purpose algorithm for monitoring Web information sources, targeted for use in conjunction with continuous query systems to achieve both timeliness (i.e., freshness of information) and completeness (i.e., observing all changes to pages) with limited resources, while showing that these two parameters are at odds and one should be compromised against the other. Our work differs from theirs in three main aspects. First, we provide a more refined definition of system constraints, partitioning updates into a request phase and processing phase. Also, our notion of completeness is different, as we are interested more in timeliness than in completeness, yet we cannot avoid repeated probing (as their model would suggest) for fear of missing a death time. Finally, we propose a concrete method for updating the stochastic model using feedback.

Much work has been done recently towards better understanding of the dynamics of web pages, their changing patterns, the lifespan of web pages and other pertinent valuable information. Pitkow [19] summarizes, based on previous surveys, statistics related to such diverse areas as web requested file popularity distributions, file sizes, site popularity, lifespan of documents, session timeouts and more. Specifically, he notes that the lifespan of
documents on the web is limited and the typical longevity for HTML pages is around 50 days, while images and other media have longer lifespans.

Cho and Garcia-Molina ([8]), as previously mentioned, conducted a large experiment involving over half a million web pages over 4 months, using their WebBase crawler (an earlier version of which was used in Google [7]) in order to estimate how web pages evolve over time. Besides the update pattern of web pages, which they found to follow Poisson processes, they also monitored the lifespan of web pages (which they defined as the period of time the pages were accessible within their time window regardless of whether the contents of the pages had changed). They found that the lifespan of web pages could be as short as less than a week (10% overall and maximum of 15% in the .com domain), and as long as over 4 months (30% overall and a maximum of 50% for the .gov domain).

Fetterly et al. [11] expanded the study done by Cho and Garcia-Molina in [8] in terms of coverage and sensitivity to change, crawling a million and a half HTML pages for 11 weeks aimed at measuring the rate and degree of web page changes. Their results verified the ones in [8] showing a strong relationship between the top-level domain (i.e. .com, .edu, etc.) and the frequency of change of a document, whereas the relationship between the top-level domain and the degree of change was found to be weaker. They also found a correlation between document size and both the frequency and degree of changes in web pages. However, the topic of web pages lifespan was not addressed in their work.
Chapter 3

Model Description

Our model consists of a mediator $M$ and a set of servers $S$. Each of the servers $s \in S$ contains a set of resources, denoted by $s.R$. The mediator is logically connected to the servers and is capable of querying each of them.

Figure 3.1 gives a graphical illustration of the model.

![Figure 3.1: the servers’ monitoring model](image-url)
The mediator $M$ has a priori estimates of the lifespan distributions of resources. These estimates are of course prone to significant error, and we wish our algorithm to withstand such errors since exact knowledge is not reasonable to attain. $M$ has a computational cycle of length $T$, and in each such cycle, the first $C$ time units are allocated for probing-related work (we assume the rest of the time is allocated for other tasks such as answering client queries, measuring server latencies, etc.) as depicted in Figure 3.2.

![Figure 3.2: the mediator’s working cycle](image)

Working in cycles, as described above, is part of the constraints the system sets. Before accessing a particular server $s \in S$, $M$ first creates a TCP connection to it, which is assumed to be a non negligible time consuming operation. Only a limited pool of TCP connections is available for the mediator, so if before connecting $s$ all connections are active, one must be broken. In such a situation, we used the least recently used (LRU) scheme for choosing which of the connections to break.

Once a TCP connection is created between the mediator and a server, the mediator can poll the resources in the server indefinitely. Specifically, probing resources in the server is performed in an asynchronous manner, i.e. first the server is sent an HTTP head request after which the mediator is free to continue without waiting for the reply to be received. Once the reply is received, it is appended to a reply queue maintained by the mediator. Replies from this queue are processed in FIFO order. We assume both request sending operations as well as reply processing operations are time consuming. We assume that each server has a typical response latency denoted by $serverlat$ which describes the time required for receiving a response from the server (i.e. the estimated time between issuing a
request and getting a reply). We also assume that serverlat values for the various servers are measured and gathered by a separate process and are readily available at the mediator’s disposal.

We have divided the HTTP replies’ status into three categories which represent the main type of replies relevant to our needs:

- **OK** - this status represents the case where the head request for the resource was successfully fulfilled. The HTTP equivalents are the replies of class 2XX, and the various redirection codes in class 3XX.

- **ERROR** - this status represents the case where the head request was unsuccessfull in the sense that it gives us no indication as to whether the resource is still alive or dead. In HTTP this category is mainly dominated by the 5XX error codes (which indicate cases where the server is aware that it has erred or is incapable of performing the request) and some error codes in 4XX which deal with client side errors.

These include the following familiar error codes:

- 404 (Not Found), where the server does not find anything matching the request. However, no indication is given as to whether the condition is temporary or permanent (contrary to 410 which will be discussed later under DEAD). The 404 code is also used is situations in which the server wishes to conceal the exact reason for the request refusal or when no other response is applicable. Note that in practice, this code is prevalently used to denote dead pages. Therefore, the proper classification of this error code depends on whether one follows its formal syntax or common pratice. We have chosen to follow the former.
– 500 (Internal Server Error), where the server encounters an unexpected condition hindering it from completing its request.

– 503 (Service Unavailable), where the server is currently unable to handle the request due to temporary overloading or maintenance of the server.

– 504 (Gateway Timeout), where the server - while functioning as a gateway or a proxy - did not receive a timely response from the upstream server or some other auxiliary server (e.g. a DNS) it needed to access in attempting to complete the request.

- DEAD - this status represents the case where the resource has expired. The HTTP equivalent is status code 410 (Gone), which according to [13] indicates that the server knows - through some internally configurable mechanism - that the resource is no longer available and no forwarding address is known. Also, this condition is expected to be considered permanent.

---

![serverAvailability pattern example](image)

Figure 3.3: serverAvailability pattern example

We have modeled a one-week failure (i.e. the chance to receive an ERROR reply mes-
sage) pattern for servers, an example of which is depicted in Figure 3.3, which shows the probability of not receiving an ERROR reply as a function of the hour of day during weekdays; for weekends we used a constant value. According to the above replies’ descriptions, we assume that each HTTP request is answered by the server, since the event that the server crashes results in a timeout which is considered to be part of the above ERROR message.

Each resource in the system has an inherent popularity, denoted by $pop$, ranging from 0 to 100 denoting minimum to maximum popularity, respectively. We assume that the popularity values for the resources are collected by a separate process and are readily available at the mediator’s disposal. The popularity of the resource reflects the importance of accurately estimating the death time of the resources.

The lifespan of a resource, or equivalently its death time, denoted by $dt$, is the time after which it dies and becomes permanently unavailable. Note that a resource can also have periodic down times, for example when the server on which it resides is temporarily unavailable (scheduled or unscheduled maintenance). Based on [19], which provides a summary of WWW characterization, we classify resources to types such as documents, images, sound, etc. and associate each type with common $dt$ estimation properties (mean and standard deviation). These estimates are later used in the algorithms we develop.

The purpose of our algorithms is to accurately estimate the lifespan of the resources. Formally, let

$$F = \sum_{s \in S} \sum_{r \in s.R} pop_r \cdot |\hat{dt}_r - dt_r|$$

where

- $S$ is the set of all servers in the system;
• \( s.R \) is the set of resources belonging to server \( s \in S \);

• \( \text{pop}_r \) is the popularity of resource \( r \);

• \( \hat{dt}_r \) is the estimated death time of resource \( r \);

• \( dt_r \) is the actual death time of resource \( r \).

We aim at minimizing \( F \), denoted hereafter as the target-function.

Note that if the algorithm was always accurate, \( F \) would be 0. Also, False positive results (the case where \( dt_i < \hat{dt}_i \)) have the same impact on \( F \) as false negative results (the case where \( \hat{dt}_i < dt_i \)), since we find \( FP \) and \( FN \) equally poor in terms of quality of service to the client.

The target-function we have formulated is one amongst various alternatives that exist. For example, a possible formulation for a target function which gives higher importance to the minimization of false positives over false negatives is

\[
F = \sum_{s \in S} \sum_{r \in s.R} \text{pop}_r \cdot w(\hat{dt}_r - dt_r) |\hat{dt}_r - dt_r|
\]

Where \( w(x) \) is weighting function that returns 0.7 if \( x \) is positive, and 0.3 otherwise.
Chapter 4

Comparative Framework for Resource Lifespan Estimation

In this chapter we establish a framework for comparing various algorithms that estimate resources’ lifespans. Let us denoted by $W$ the (potentially infinite) set of all such algorithms. $W$ can be logically partitioned as follows:

- $W_{np}$ — all algorithms that do not probe the resources, i.e. all estimates regarding the death times of the resources are made without any feedback from the resources whatsoever.

- $W_p$ — all algorithms in $W - W_{np}$.

Let us first take a closer look at $W_{np}$. Since the estimation of the deaths times of resources is based solely on the a priori death time distributions known to the mediator, we expect the results to be poor. This is validate by the simulation we conduct as reported in Section 6.2.

Thus, we now focus on $W_p$, which we next partition further according to the following criteria:
• **The validation (probing) scheme** - knowing that all algorithms in $W_p$ perform probing to resources we distinguish between those that employ a simple circular round-robin validation scheme, denoted by *Uniform*, and all others, denoted by *NonUniform*, since they adapt the probing frequency according to various criteria.

An orthogonal distinction is among *batched* and *non-batched* validation schemes: in a *batched* validation schemes the resources from each server are probed in batches in specific working cycles. Formally, if $r^x_i$ denotes resource $i$ belonging to server $x$, then examining a sequence of probings in a given cycle $r^{s_1}_n, r^{s_2}_m, \ldots r^{s_n}_m$ will satisfy

$$\forall i \in 1 \ldots n - 1, \text{ if } \exists j \text{ s.t. } j > i \text{ and } s_i = s_j \text{ then } \forall k \in i \ldots j, s_k = s_j$$

*Non-batched* validation schemes are those that do not follow the above rule.

• **Resources’ life estimation** - Regardless of the validation scheme, once a resource is probed the algorithm at hand must estimate whether the resource has expired or not. If it estimates the resource has not yet expired, a future probing will be scheduled. Thus, we assume all algorithms have an estimate of the existence probability of each resource, which we denote by $Ep$, and these estimates are updated each time the relevant resources are probed. Note that we could have equally chosen to estimate the death probability of each resource, which would simply be $1 - Ep$.

We distinguish between two algorithmic categories for calculating $Ep$:

- **$EpDist$**: $Ep$ is updated according to the estimated distribution of each resource only (recall that our model assumes the mediator has a priori estimates of the lifespan distribution of the various resources, based on their type characteristics). Each time a reply is processed, $Ep$ is calculated based on the estimated
mean and standard deviation of the normal distribution of the lifespan of a resource of the given type. If this probability is below a certain threshold, the resource is announced dead. Otherwise, a future probing of the resource will take place according to the validation scheme employed. Note that although the calculation of \( E_p \) is based only on the a priori estimated distributions, probing is still essential because of the possibility to receive a \texttt{DEAD} status code reply, in which case the resource is announced as dead immediately. Also, when \texttt{OK} is received, the resource will not be announced dead, no matter its \( E_p \) value (\( E_p \) still has to be calculated though since its value may have impact on the validation algorithm employed).

– \textit{EpCombined}: \( E_p \) is not updated solely based on the a priori lifespan estimates but incorporates other factors as well. We develop a scheme of \( E_p \) calculation that incorporates both historical information of past probe replies along with estimated normal distributions. We discuss this scheme elaborately in Section 5.2.

A graphical summary of the above classification is given in Figure 4.1. The \( x - y \) plain chooses among the different possible validation schemes where the \( x \) axis chooses the frequency of validation scheme, i.e. whether the algorithm is a uniform or a non-uniform one, and the \( y \) axis chooses among the batched schemes and the non-batched ones. The \( z \) dimension chooses among the two proposed \( E_p \) calculation categories.

We shall focus from now on only on algorithms that belong to the \textit{batched} group (the non-shaded box in Figure 4.1) for two reasons: First, the simulations we have conducted show indeed that as we increase the ratio between the number of servers to the the size of the TCP connection pool the results achieved by \textit{non-batched} algorithms deteriorate. Second, recall that our model requires the establishment of a TCP connection, which is an
expensive operation, to the server in which the resource to be probed resides. Also, our model assumes a limited pool of available TCP connections. Given a sequence of probing to perform on resources residing in different servers, it is obvious that the time needed to complete the sequence would be shorter had we used a batched algorithm, in case the number of different servers in the sequence is bigger than the size of the TCP connection pool.

For instance, suppose a probe costs 10[ms] and a TCP connection creation costs 30[ms]. Also, assume that the TCP connection pool available for the mediator is a single connection.

Let the sequence of probes to perform be

\[ r_1^1, r_2^2, r_3^1, r_4^2, r_5^1 \]
where upper indices denote the server number and lower indices denote a resource number. A non-batched algorithm would probe the sequence as it is and would cost \((10 + 30) \cdot 5 = 200 \text{ [ms]}\) since we have 5 probes and each probe requires the creation of a new TCP connection. That stems from the fact that the pool is of size 1 and therefore before each probe we have to break the previous connection for the sake of creating the new one since the servers always change from one probe to another.

However a batched algorithm would batch the probes according to the servers yielding possibly the sequence

\[
r_1^1, r_3^1, r_5^1, r_2^2, r_4^2.
\]

Note that this sequence would cost only \(10 \cdot 5 + 2 \cdot 30 = 110 \text{ [ms]}\) since now only 2 TCP connections have to be established: one before \(r_1^1\) is probed and the other before \(r_2^2\) is probed.

Therefore, in order to keep our discussion concise we shall focus on the reduced algorithms’ classification matrix presented in Figure 4.2, which results from the eliminated of the y axis in Figure 4.1.

We now discuss details of representative algorithms for the more trivial slots in the matrix in Figure 4.2, namely those which use a validation scheme belonging to Uniform with an Ep calculation scheme belonging to EpDist. The remaining slots are the subject of the next chapter. Since the two schemes are orthogonal, we discuss each separately.

\section{4.1 The Uniform Validation Scheme}

A Uniform validation scheme simply probes all resources circularly resulting in uniform probing frequency to all resources. Being a batched algorithm, there is no interleaving of
resource probings belonging to different servers, i.e. each time a server is visited, all of its scheduled resources are probed.

The algorithm maintains for each resource an estimate of its $Ep$ which is updated according to the employed $Ep$ calculation scheme (i.e. either $EpDist$ or $EpCombined$). Note that this algorithm induces a constant probing rate for all resources.

Also, neither the order in which the servers are visited nor the order in which the resources are visited in each server is of any significance. A concise representation of this algorithm is given in Figure 4.3. The algorithm uses the following data structures and global variables:

- $S$ - a linked list of all the servers in the system (the order of the servers in the list is arbitrary).

- $curS$ - a pointer to the active server, i.e. the server from which the next resource
will be probed.

- \( s.R \) - the list of resources in server \( s \), for each \( s \in S \) (the order of the resources in the list is arbitrary).

- \( \text{curR} \) - a pointer to the next resource to be probed.

We also assume that an iterator-like data structure can be invoked on \( S \) and on any of the resource lists.

Given a linked list \( L \), the functions used by the iterator to manipulate \( L \) are:

- \( \text{open}(L) \) - initializes the iterator without returning any item.

- \( \text{getNext}(L) \) - returns the next item in the list and adjusts the internal data structure of the iterator needed to get the next items in subsequent calls. \text{null} is returned in case there are no more elements.

- \( \text{hasNext}(L) \) - returns \text{true} if the end of the list has not been reached. Otherwise, it returns \text{false}.

- \( \text{close}(L) \) - closes the iterator and deletes the internal data structure supporting it.

- \( \text{getFirst}(L) \) - returns the first item in the list.

We assume that a call to \text{Initialize} given in Figure ?? has been made prior to running the algorithm.

### 4.2 The EpDist \( \mathcal{E}_p \) Calculation Scheme

The \( \mathcal{E}_p \) calculation scheme estimates the \( \mathcal{E}_p \) parameter for each resource based solely on the a priori estimates of the lifespan distribution of the resources held by the mediator.
while ( end of current cycle not reached )

1. if TCP connection not established to \textit{curS}
   \begin{itemize}
   \item if TCP connection pool is full
     \begin{itemize}
     \item close connection to a server according to LRU policy
     \end{itemize}
   \item establish a TCP connection to server \textit{curS}
   \end{itemize}
2. send a head HTTP request to \textit{curR}
3. check the reply queue for a pending reply and if one exists, process it
4. if \texttt{hasNext(curS.R)} = \texttt{false} then
   \begin{itemize}
   \item close\texttt{(curS.R)}
   \item if \texttt{hasNext(S)}=\texttt{false} then \texttt{curS=getFirst(S)} else \texttt{curS=getNext(S)}
   \item open\texttt{(curS.R)}
   \end{itemize}
5. \texttt{curR=getNext(curS.R)}

Figure 4.3: a \textit{Uniform} validation scheme

- \texttt{curS = open(S)}
- \texttt{curR= open(curS.R)}

Figure 4.4: \textbf{Initialize} - initialization of the \textit{Uniform} validation scheme

Figure 4.5 describes the \textit{EpDist} calculation scheme. This scheme is applied after a resource is probed and its reply is being processed.

We assume the mediator maintains a priori lifespan distributions in the array \texttt{lifeSpanEstimates} which is indexed by the type of the resource (assumed to be an integer value). The type of the resource is retrieved by the function \texttt{getType()} which is applied to a resource. The \textit{Ep} field of a resource \texttt{r} is accessed by \texttt{r.Ep}.

\textbf{EpDist} first recalculates the resource’s \textit{Ep} field, based on \texttt{lifeSpanEstimates}. Then, based on the updated value of \textit{Ep} and the status of the reply it decides whether the resource should be announced as dead or not: if the reply status is \texttt{DEAD}, it kills the resources. However, when it is \texttt{ERROR}, it checks up the \textit{Ep} field of the resource, and if \textit{Ep} < \texttt{threshold}, it announces the resource as dead. \texttt{threshold} is a tunable parameter we further discuss in
• Let $r$ be the resource whose reply is processed.
• $type = \text{getType}(r)$
• calculate $r.Ep$ according to the distribution in $\text{lifSpanEstimates}[type]$
• Check the status of the reply
• if $replyStatus = \text{DEAD}$ or $(r.Ep < \text{threshold}$ and $replyStatus = \text{ERROR}$) then announce resource as dead

Figure 4.5: $EpDist$ calculation scheme

Chapter 6. Note that if the reply status is OK no action is taken, i.e. the resource will be regarded as alive and will continue to be probed in the future.
Chapter 5

The NonUniform and EpCombined Algorithms

In this chapter we discuss the NonUniform and EpCombined algorithms. Recall from Chapter 4 that NonUniform is a validation scheme and EpCombined is an Ep calculation scheme and therefore the description of each is independent of the other. Moreover, an NonUniform validation scheme can be used with either Ep calculation algorithm, and EpCombined can be used with either validation scheme. We first discuss the NonUniform algorithm and then turn our attention to EpCombined.

5.1 The NonUniform Validation Algorithm

In the previous chapter we discussed the Uniform validation algorithm and saw that in Uniform all resources are probed at the exact same frequency. In NonUniform, however, we adapt the probing frequency based on Ep and Popularity parameters of the resources.

We start by describing the algorithm’s data structures along with the accompanying
notation, and afterwards describe the algorithm itself with its analysis.

## 5.1.1 Data Structures

Each resource in the system is assigned with three fields that *dynamically change* along the course of the algorithm’s execution:

- **age** - the age field is an integer value, initially set to zero, which is introduced to prevent *starvation* of a resource. Starvation means the absence of probing to a scheduled resource. The exact manipulation of the *age* field is described in Section 5.1.2. Basically, if a resource is scheduled to be probed at some cycle and is not probed, then we increase its age. Once probed, the age is reset to 0.

- **Ep** - the existence probability of a resource. This is a dynamic parameter which is initially set to 100% for all resources and is updated according to the *Ep* calculation.

- **rank** - the *rank* of a resource encapsulates how important it is for us to probe the resource frequently and hence determines the frequency of the resource probing. The rank of the resource is determined by two factors. First, the popularity of the resource, i.e. its *pop* field (which is assumed to be constant for each resource). Second, by its *Ep* field described above. The exact function relating *Ep* and *pop* to *rank* is given in Section 6.3.

*NonUniform* uses a sliding window of size *MAXQ*, which we denote as the *epoch*. *MAXQ* is a configurable system parameter (the setting of which is discussed later in Section 5.1.4). Each item, or portion in the epoch represents a working cycle of *C* time units. As mentioned in Chapter 3, the time between one working cycle and the next is *T* time units, where *C* < *T*.

For each server *s* ∈ *S* in the system the mediator maintains:
• A window, \( RQ \), with \( MAXQ \) buckets, where bucket \( RQ[i] \) (bucket \( i \) in \( RQ \)) holds a \textbf{min-max heap} \[2\] of all resources that are scheduled to be probed in the \( i \)-th cycle. That is, \( RQ[0] \) holds resources to be probed in the current cycle, \( RQ[1] \) holds resources to be probed in the next cycle, etc. The key for the min-max heap organization is the ordered pair \((age, rank)\) of each resource, where keys are compared lexicographically. As shown in [2], a \textbf{min-max heap} is a heap-like data structure supporting access to the minimum and maximum valued key elements in \( O(1) \) time, and insertions, deletion of the maximum/minimum element in \( O(\log n) \), \( n \) being the number of elements in the heap.

We have used a min-max heap since we wish to probe resources in non-increasing order of \((age, rank)\) (thus need to extract maximum \((age, rank)\) elements), and as we show in Section 5.1.3 - access to the minimum \((age, rank)\) valued resource in \( RQ \) is also necessary for the algorithm. An alternative to using min-max heaps could be the usage of two heaps: minimum and maximum with the linkage of identical elements in the heaps. This, however, would lead to an increase in space complexity.

• An array of pairs of integers denoted by \textit{weight}, s.t.

\[
\forall i \in 0\ldots MAXQ - 1, \text{weight}[i] = \sum_{r \in RQ[i]} (r.age, r.rank).
\]

Thus, the \textit{weight} field in each bucket calculates the accumulated \textit{age} and \textit{rank} of all resources scheduled in the current bucket at the given server. The \textit{weight} array is updated upon insertions/ deletions in the relevant \( RQ \) buckets.

In addition, the mediator holds a global window, \( SC \), which also contains \( MAXQ \) buckets, where bucket \( i \) holds the following information regarding \( i \) cycles \textit{ahead} (when \( i = 0 \) it refers to the current cycle):
Figure 5.1: data structures of *NonUniform*
• $ns$ - an estimate of the number of HTTP head requests to send in this cycle.

• $np$ - an estimate of the number of replies to process in this cycle.

• $minR$ - a minimum heap containing the resources with minimum $(age, rank)$ in $RQ[i]$s of all the servers. Formally, the resources contained are

$$\{\forall s \in S \min(s.RQ[i])\}$$

where $S$ is the set of servers in the system, and the function $\min$ extracts the minimum resource from the minimum heap it is applied to.

• $SH$ - a maximum heap containing the servers with the key being the $weight[i]$ field of the servers.

Thus, in each cycle $i$ the algorithm estimates the expected workload and performs a reference - in $O(1)$ time - to the resource with the minimum $(age, rank)$ pair amongst all resources scheduled in $i$. This estimation will later be used in order to schedule future resource probings.

The HTTP replies received from the servers in reply to the head requests are queued in a FIFO reply queue denoted as $RepQ$.

Figure 5.1 gives a graphical representation of the above.

5.1.2 Main Loop of NonUniform

An outline of the algorithm’s main running loop is described in Figure 5.2. We next elaborate on the details of each step of the algorithm.
1. while (end of current cycle not reached yet)
   (a) choose the server, \(s\), to probe
   (b) choose the resource, \(r\) (\(r \in s\))
   (c) send a head HTTP request to \(r\)
   (d) estimate the reply arrival time from server \(s\) (according to \(s.serverlat\)) and update \(SC\)’s relevant bucket accordingly (i.e. increment \(np\) of the relevant bucket by 1)
   (e) check \(repQ\) for a pending reply, and if one exists:
      • let the subject of the reply be resource \(r\)
      • apply the chosen \(Ep\) calculation scheme on the reply
      • if \(r\) not announced dead
         – set \(r.age = 0\)
         – reschedule \(r\)’s future probe

2. evacuate resources left to be probed in the current cycle by re-scheduling them in future cycles.

Figure 5.2: main loop of \textit{NonUniform}

Choosing the resource to probe

In Figure 5.2 we see that the task of choosing the resource to probe is preceded by choosing the server to probe. Recall that we concentrate on \textit{batched} algorithms, and as such, \textit{NonUniform} first chooses a server to probe, establishes a TCP connection to it (if one is not established already), and probes \textit{all} scheduled resources in its current bucket.

To that end, each time a new cycle starts we initialize according to Figure 5.3.

- Retrieve the \(weight[i]\) fields of all servers
- Build a maximum heap, \(SH\), of all servers keyed by the servers’ \(weight[i]\) field
- initialize a pointer to the chosen server \(serverChosen = null\)

Figure 5.3: initialization at start of cycle \(i\)

Figure 5.4 describes the server selection routine. We have used \textbf{extract-max} as a
if $serverChosen = \text{null}$ then
  – if $\text{is-empty}(SH)$ then exit /*all scheduled probing for the current cycle done*/
  – else $serverChosen = \text{extract-max}(SH)$

return $serverChosen$

Figure 5.4: select server in NonUniform

routine that extracts the element with maximum key ($weight$ field in our case) from the
heap, and $\text{is-empty}$ as a routine that checks whether the heap is empty. Once a server is
chosen from the heap $SH$, all the scheduled resources for the current cycle in that server
are probed. When done the field $serverChosen$ is reset to $\text{null}$ (see Figure 5.5) and the
next server is then chosen from the heap $SH$, and so on.

The complexity of building the heap is $O(S)$, where $S$ is the number of servers,
whereas the complexity of choosing the next server from the heap is $O(\log S)$.

We now turn to the resource selection routine. As previously mentioned in Sec-
tion 5.1.1, for each server $s \in S$, $s.RQ$’s buckets, $s.RQ[0] \ldots s.RQ[\text{MAXQ} - 1]$, are
$\text{min-max}$ heaps holding the resources to be scheduled in each bucket, where the heap key
is the ordered pair $(age, rank)$ of the resources.

The resource selection routine, given in Figure 5.5 uses the $serverChosen$ variable
given in 5.4, assuming it is not $\text{null}$ (this assumption is valid based on the initial condition
in Figure 5.4. Recall that each call to the select resource routine is preceded by a call to
select server (see Figure 5.2).

In line 2 of 5.5, we set $serverChosen$ to $\text{null}$ when we reach the last resource in the
current server. Thus, the next time the select server routine is called, a new server will
be elected from the heap $SH$. Note that we assume each server in $SH$ has at least one
1. Let the current bucket index be \( c \).

2. if \( \text{size}(\text{serverChosen.RQ}[c]) \leq 1 \) then set \( \text{serverChosen} = \text{null} \)

3. return \( \text{extract-max}(\text{serverChosen.RQ}[c]) \)

Figure 5.5: select resource in NonUniform

resource scheduled. The complexity of select resource is \( O(\log(\text{size}(RQ[c]))) \), where \( \text{size} \) is the size of the heap and \( c \) is the index of the current heap.

5.1.3 Re-scheduling Future Resource Probing

In Figure 5.2 we saw that in case the chosen \( Ep \) calculation scheme does not render the resource as dead, a re-scheduling of its future probing is performed. In this section we discuss the re-scheduling scheme employed in NonUniform.

First, let us recall the data structure \( SC \) initially mentioned in Section 5.1.1. This is a window of size \( MAXQ \), where each bucket holds information that estimates the workload anticipated in the current cycle, the next cycle, etc., up to \( MAXQ - 1 \) cycles ahead.

**Definition 5.1.** The cost of a bucket \( b \) in \( SC \) (\( SC[b] \)), referred to as \( \text{cost}(b) \) is defined as \( SC[b].ns \cdot ts + SC[b].np \cdot tp \), i.e. the amount of time needed to complete the estimated workload of the bucket.

Recall that \( ts \) and \( tp \) are the estimates of the time required to send a request and process a reply, respectively, whereas \( ns \) and \( np \) are the estimates of the number of requests to send and replies to process in a single cycle, respectively.

**Definition 5.2.** \( SC[b] \) is said to be full if \( \text{cost}(b) \geq C \)

The rank field of the resource is the parameter that, along with the age field, determines where the next probing of the resource will take place.
Section 5.1.4 discusses how the rank of the resource is calculated based on its $Ep$ and $pop$ fields. Once the rank is calculated, we map it to the next-cycle in which we schedule its future probe. This mapping is done by a heuristically chosen linearly decreasing function (using the principle of Occam’s razor) denoted as $\text{rank2nextCycle}$. 

$$rank2nextCycle : 0 \ldots 100 \rightarrow 1 \ldots MAXQ - 1$$

Specifically, next-cycle decreases from $MAXQ - 1$ to 1 as the rank increases from 0 (minimum rank) to 100 (maximum rank), respectively. Thus, when the rank is minimum, the next probe of the resource will be scheduled $MAXQ - 1$ cycles ahead. When the rank reaches its maximum value, the next probe will be scheduled 1 cycle ahead, i.e. in the next cycle.

**Definition 5.3.** The lateness of a schedule of a resource $r$ is the number of cycles after $\text{Rank2NextCycle}(r.rank)$, in which $r$ is scheduled ($r.rank$ denotes the rank).

**Definition 5.4.** The weighted-lateness of a schedule of a resource $r$ is its lateness multiplied by its rank.

The objective of the re-scheduling algorithm is to minimize weighted-lateness under the server processing constraint.

Figure 5.6 outlines the re-scheduling algorithm. It assumes that the rank of the resource has been calculated, and that $i$ is the resulting cycle after the application of the $\text{rank2nextCycle}$ function to rank.

re-schedule uses the auxiliary function schedule, presented in Figure 5.7.

In Figure 5.7 we use the following macros to work on a doubly linked list $B$: 33
• let \( i \) be the cycle of interest, and \( c \) be the current cycle

• let \( s \) be the server to which \( r \) belongs

• create a linked list \( B \) of items representing the buckets \( b_0, b_1, b_2 \ldots b_{\text{MAXQ}-1} \) where 
  \( b_0 \) represents the current bucket, \( b_1 \) is the following bucket, etc

• set the index field of each item in \( B \) to the index of the bucket in \( SC \) to which it points:
  \[
  \forall k \in [0 \ldots \text{MAXQ} - 1] \ b_k.index = (c + k)\%\text{MAXQ}
  \]

• \( \text{schedule}(r, b_{(i-C)\%\text{MAXQ}}, B) \)

Figure 5.6: re-scheduling algorithm \( \text{re-schedule}(r, i) \)

• each item in the list \( b \in B \) is a structure holding the index of a bucket in \( SC \) in its
  index field, \( b.index \). Thus, items in \( B \) can be regarded as pointers to buckets in \( SC \).

• \( \text{first}(B) \) - returns the first item in the list or \( NULL \) if the list is empty.

• \( B - b \) - returns the list resulting from the removal of item \( b \in B \) from it.

• \( \text{last}(B) \) - returns the last item in the list or \( NULL \) if the list is empty.

• For each item \( b \in B \) in the list we have:
  – \( \text{prev}(b) \) - returns the item in \( B \) that precedes \( b \) or \( NULL \) if \( b \) is the first item in
    the list.
  – \( \text{next}(b) \) - returns the item in \( B \) that follows \( b \) or \( NULL \) if \( b \) is the last item in
    the list.
  – \( \text{closest}(b) \) - returns \( \text{next}(b) \) if \( b \) is not the last item in the list. Otherwise, it
    returns \( \text{prev}(b) \).

\( \text{schedule} \) uses the following variables:

• \( r \) - the resource to schedule.
- $B$ - a doubly linked list of items pointing to buckets in $SC$ with increasing bucket index with $\text{first}(B) . index \geq 0$ and $\text{last}(B) . index \leq MAXQ - 1$, and $b \in B$ points to the bucket to which $r$ is scheduled.

- $sign$ - indicates whether the next bucket to schedule is after $b$ (1) or before $b$ (-1) denoted by $forward$ and $backward$ buckets, respectively. $b$ itself is considered a forward bucket by convention.

- $oneDir$ - a boolean variable indicating whether $sign$ should be changed in the next iteration (FALSE) or not (TRUE).

- $b_c$ - the next bucket in $B$ to check.

- $b_f$ - the next $forward$ bucket to check.

- $b_b$ - the next $backward$ bucket to check.

In addition, $schedule$ uses two auxiliary routines, namely $insert$ and $remove$, described in figures 5.8 and 5.9, respectively.

$schedule$ is a recursive routine. Its objective is to $insert$ $r$ to bucket $b$. If the bucket is not full, it does so (part 2(d)) and quits.

If the bucket is full then let $m$ be the resource in bucket $b_c$ with the minimum $(age, rank)$ pair:

1. If $(age, rank)$ of $r$ is bigger then $(age, rank)$ of $m$ (where lexicographic comparison is done among the pairs), then $m$ is removed from $b_c$, giving precedence to $r$ which is then inserted to $b_c$ (part 2(f)) and a recursive call to $schedule$ is made with $m$ as the first parameter. As we shall show, bucket $b_c$ can be removed from the list of legitimate buckets in which we should try to schedule $m$, and so we use the macro
1. set $\text{succ} = \text{false}$, $\text{sign} = 1$, $\text{bc} = \text{null}$, $\text{oneDir} = \text{false}$, $\text{bf} = b$, $\text{bb} = \text{prev}(b)$

2. while $\text{succ} = \text{false}$ /*flagging successful scheduling of $r$ */
   
   (a) set $\text{bc} = \text{null}$
   
   (b) while $\text{bc} = \text{null}$ and not ($\text{bf} = \text{null}$ and $\text{bb} = \text{null}$)
       
       - if $\text{sign} == 1$
         
         - if not $\text{bf} = \text{null}$ then set $\text{bc} = \text{bf}$, $\text{bf} = \text{next}(\text{bf})$
         
         - else set $\text{sign} = -1$, $\text{oneDir} = \text{true}$
       
       - if $\text{sign} == -1$
         
         - if not $\text{bb} = \text{null}$ then set $\text{bc} = \text{bb}$, $\text{bb} = \text{prev}(\text{bb})$
         
         - else set $\text{sign} = 1$, $\text{oneDir} = \text{true}$
       
       - if $\text{oneDir} = \text{false}$ then set $\text{sign} = \text{sign} \cdot -1$

   (c) if $\text{bc} = \text{null}$ exit while /* no bucket left to check */

   (d) if bucket $\text{bc}$ is not full
       
       - insert($r, \text{bc}.index$)
       
       - set $\text{succ} = \text{true}$; exit while

   (e) Let $m$ be the resource in bucket $\text{bc}$ with the minimum $(\text{age}, \text{rank})$ pair.

   (f) if ($\text{r.age + bc.index} - \text{b.index}, \text{r.rank}) > (\text{m.age}, \text{m.rank})$ then
       
       - set $\text{r.age+ = bc.index - b.index}$, $\text{succ} = \text{true}$
       
       - remove($m, \text{bc.index}$), insert($r, \text{bc.index}$)
       
       - set $\text{bm} = \text{closest}(\text{bc})$, $\text{m.age+ = bm.index - bc.index}$
       
       - schedule($m, \text{bm}, B - \text{bc}$)

   (g) else $B = B - \text{bc}$

3. if $\text{succ} = \text{false}$
   
   - insert($r, m$), where $m$ is the index of the bucket with minimum cost closest to $b$

Figure 5.7: $\text{schedule}(r, b, B)$

closest to find the first valid optional bucket to try to insert $m$ to, denoted by $b_m$, and call recursively $\text{schedule}$ with $b_m$, while $b_c$ is subtracted from the list of buckets $B$.

Note that the age of $m$ is modified according to the selection of $b_m$.

2. If $(\text{age}, \text{rank})$ of $r$ is not bigger then $(\text{age}, \text{rank})$ of $m$, then the next valid bucket from $B$ is inspected, as done in the previous bullet with the appropriate modification of the age field of $r$ as follows:

Let $b_c$ be the next legitimate bucket chosen from $B$ (in the next paragraph we discuss
input:  
• $r$ - the resource
• $i$ - the bucket

actions:  
• let $s$ be the server to which $r$ belongs
• add $r$ to $s.RQ[i]$
• set $SC[i].ns \leftarrow SC[i].ns + 1$
• update $minR[i]$ if necessary

Figure 5.8: $\text{insert}(r, i)$

input:  
• $r$ - the resource to remove
• $i$ - the bucket to remove from

actions:  
• let $s$ be the server to which $r$ belongs
• remove $r$ from $s.RQ[i]$
• set $SC[i].ns \leftarrow SC[i].ns - 1$
• update $minR[i]$ if necessary

Figure 5.9: $\text{remove}(r, i)$

its selection procedure). Then, we compare $(age + b_c.index - b.index, r.rank)$ of $r$ instead of $(age, rank)$ to the minimum $(age, rank)$ resource $m$ in $b_c$.

The buckets from $B$ are chosen in a toggling two direction linear manner, i.e. first bucket $b$ is inspected for insertion, as explained above. If the required conditions are not met for $r$’s insertion (bucket full or minimum resource in bucket has greater $(age, rank)$ then $r$’s) then the next backward bucket from $B$ is checked. If again the conditions are not met then the next forward bucket is inspected and so forth, toggling between forward and backward buckets, with the appropriate modification of $r$’s $age$ field. For example, in Figure 5.10, if $b = 3$ and the current bucket is $c = 1$ then the order of bucket checking would be 3,2,4,1,5. In case one direction
finishes, then we of course continue in the remaining direction without toggling, e.g. if \( b = 4 \) then the order would be 4,3,5,2,1. The above is accomplished in part 2(b) of Figure 5.7.

![Diagram showing bucket choosing in list B]

Figure 5.10: illustration of the bucket choosing in list \( B \)

If \( r \) is not inserted to any of the buckets in \( B \) (which could occur if all buckets in \( B \) are full, and \( r \) has the least \((age, rank)\) pair in all buckets) then the bucket chosen for insertion is the one with minimum cost (in case of ties, the bucket closest to \( B \) is chosen). This is done in order to equally distribute the load in each of the buckets.

Figure 5.11 describes an example run of the rescheduling algorithm. Suppose the list of resources to be scheduled is \((4, 4), (4, 3), (4, 2)\) and \((4, 1)\) where \((i, j)\) denotes the \((age, rank)\) pair of the resource. Also, suppose all resources are to be scheduled to bucket 3 and each bucket can allocate a single resource only.

First resource \((4, 4)\) is inserted to bucket 3 as shown in Figure 5.11(a). Next resource \((4, 3)\) is checked for insertion to bucket 3, but since the bucket is filled with \((4, 4)\) and \((4, 3) < (4, 4)\), bucket 2, being the next legitimate bucket for insertion is chosen. In (b) we see that the age of the resource is decreased to 3. Next resource \((4, 2)\) checks bucket 3 for insertion. Since \((4, 2) < (4, 4)\), bucket 2 is checked with the appropriate decrement of \((4, 2)\)'s age to \((3, 2)\) because a previous bucket is checked. Since bucket 2 now contains \((3, 3)\) and \((3, 2) < (3, 3)\), the next legitimate bucket for insertion, bucket 4, is chosen.
Figure 5.11: run example of reschedule: scheduling list of resources [(4, 4), (4, 3), (4, 2), (4, 1)] to bucket 3.

The age of the resource is incremented accordingly as shown in (c). In a similar manner, resource (4, 1) is finally inserted to bucket 1, as shown in (d).

Next, we prove that the number of recursive calls to schedule is bounded by $MAXQ$. We first prove that each bucket can only be visited once in the course of the re-scheduling algorithm (Theorem 5.2 using Lemma 5.1). Finally, we prove that once a bucket is either visited or checked, it can be safely removed from the list of legitimate buckets that need to be checked for insertion (Theorem 5.5 using Lemma 5.3).

**Definition 5.5.** In schedule we say a bucket $b$ is visited when we insert $(r, b)$, and accordingly remove the resource with minimum $(age, rank)$ pair in $b$, $r_m$, which satisfies $(r.age, r.rank) > (r_m.age, r_m.rank)$.

**Lemma 5.1.** In schedule, resources removed from a specific bucket once visited, are in increasing lexicographical order of their $(age, rank)$ values.
Proof Let \( b \) be some bucket, and suppose a resource \( r_1 \) is removed, followed by a later removal of \( r_2 \). When \( r_1 \) was removed it had the minimum \((age, rank)\) pair in \( b \), by definition. Let \( r_i \) be the resource inserted to \( b \) upon \( r_1 \)’s removal. According to schedule, \((r_i.age, r_i.rank) > (r_1.age, r_1.rank)\), so it follows that upon removal of \( r_1 \), for each resource \( r \) left in \( b \), \((r.age, r.rank) > (r_1.age, r_1.rank)\). Since the next removal of a resource from \( b \) occurs from this resource population, and resources in buckets do not change their \((age, rank)\) value, it follows that \((r_2.age, r_2.rank) > (r_1.age, r_1.rank)\)

Theorem 5.2. In schedule, each bucket is visited only once.

Proof Suppose, by way of contradiction, that there is a bucket that is visited more than once. Then let \( b_1 \) be the first bucket to be visited more than once. Let \((age_1, rank_1)\) be the age and rank of the resource removed from \( b_1 \) in the first visit. Similarly, let \((age_2, rank_2)\) be the age and rank of the resource removed from the next bucket visited, \( b_2 \), and so forth. Finally, let \((age_n, rank_n)\) be the the age and rank of the resource removed after \( n \) total visits and in the second visit of \( b_1 \) (where \( b_1 = b_n \)).

According to schedule, the resource removed from \( b_1 \), is the one we insert to \( b_2 \) with an updated age value according to the distance (in buckets) between \( b_1 \) and \( b_2 \) satisfying \((age_1 + b_2 - b_1, rank_1) > (age_2, rank_2)\). The same holds for the rest of the visits which results in the following equations:

\[
(age_1 + b_2 - b_1, rank_1) > (age_2, rank_2) \\
(age_2 + b_3 - b_2, rank_2) > (age_3, rank_3) \\
(age_3 + b_4 - b_3, rank_3) > (age_4, rank_4) \\
\vdots \\
(age_{n-1} + b_1 - b_{n-1}, rank_{n-1}) > (age_n, rank_n)
\]

follows since \( b_n = b_1 \)
summing up the inequations, all the $b_i$’s cancel and we get:

$$(age_1 + \sum_{i=2}^{n-1} age_i, rank_1 + \sum_{i=2}^{n-1} rank_i) > (age_n + \sum_{i=2}^{n-1} age_i, rank_n + \sum_{i=2}^{n-1} rank_i)$$

which, upon cancellation of the common terms results in:

$$(age_1, rank_1) > (age_n, rank_n)$$

Note that the above equations relates to (age, rank) pairs of resources both removed from the same bucket $b_1$, when $r_n$’s removal comes after $r_1$. According to Lemma 5.1, the relation that follows is

$$(age_n, rank_n) > (age_1, rank_1)$$

which contradicts the above. Thus, a bucket cannot be visited more than once. ■

**Definition 5.6.** In schedule we say a bucket $b$ is checked when the resource with minimum $(age, rank)$ pair in $b$, $r_m$, satisfies $(r.age, r.rank) \leq (r_m.age, r_m.rank)$.

Note that to accomplish a check operation $O(1)$ time is needed since it involves a single comparison of $r$ to $r_m$, and $r_m$ is accessible in $O(1)$ time when the buckets are implemented in a min-max heap as specified in [2], and the global minimum heap $minR$ holds the minimum among the minimums over all the servers.

**Definition 5.7.** In schedule we define the relative age-rank pair of a resource $r$ located in bucket $b_i$ (where $i$ is the index of the bucket) by

$$rel_r = (age_r - i + 1, rank_r)$$
If $r$ is the resource to be scheduled then $b_i$ is the bucket parameter to \textit{schedule}.

\textbf{Lemma 5.3.} In \textit{schedule}, if a visit occurs in bucket $b_i$, then

\[
\text{rel}_r > \text{rel}_m
\]

where $r$ is the resource inserted to $b_i$, and $m$ is the one removed from it.

\textbf{Proof} If $r$ is the resource parameter to \textit{re-schedule}, then let $b_j$ be the second parameter to it, i.e. the initial call to \textit{re-schedule} was \textit{re-schedule}$(r,b_j)$. Otherwise, let $b_j$ be the bucket from which $r$ was removed as a consequence of a preceding visit to to $b_j$.

According to \textit{schedule}, if $r$ is inserted to $b_i$, and $m$ is removed from it, then

\[
(age_r + i - j, rank_r) > (age_m, rank_m)
\]

If we add $(1 - i, 0)$ to both sides of the inequality we get

\[
(age_r - j + 1, rank_r) > (age_m - i + 1, rank_m)
\]

which reduces by definition 5.7 to

\[
\text{rel}_r > \text{rel}_m
\]

\textbf{Corollary 5.4.} Let $r_1, r_2, \ldots, r_n$ be the list of resources for which \textit{insert} is called in \textit{schedule} ($n$ is finite and bounded from above by $MAXQ$ according to Theorem 5.2). Then the
following holds:

\[ \text{rel}_1 > \text{rel}_2 > \cdots > \text{rel}_n \]

**Proof** The proof is by induction on the length of the list \( n \).

**base** \( n = 1 \) - trivial.

**step:** Our inductive hypothesis is that after the \( k \)th visit, \( \text{rel}_1 > \text{rel}_2 > \cdots > \text{rel}_k \) and we need to prove for the \( k + 1 \)th visit that \( \text{rel}_1 > \text{rel}_2 > \cdots > \text{rel}_{k+1} \). To that end all is left to show is that \( \text{rel}_k > \text{rel}_{k+1} \). Note that when \( r_k \) is inserted, some other resource is removed from it, otherwise according to **schedule** there would be no further insertions and specifically the insertion of \( r_{k+1} \) would not take place which leads to a contradiction. Therefore, let \( r_{k+1} \) be the resource removed, since after its removal it will be the next resource inserted. According to Lemma 5.3 \( \text{rel}_k > \text{rel}_{k+1} \), which completes our inductive step.

\[ \blacksquare \]

**Definition 5.8.** In **re-schedule**, a check is said to be superfluous if it is known that the checked bucket would not be visited. (i.e. if \( m \) is the minimum \((\text{age}, \text{rank})\) paired resource in the checked bucket, \( b \), and \( r \) is the resource to schedule then it follows that \( \text{rel}_r \leq \text{rel}_m \)).

Obviously if we know a check is superfluous we would rather eliminate it. In **schedule**, this is accomplished in *part 2(g)*, where \( b \) is subtracted from the list of buckets \( B \).

This step may save us a computation time since each check costs \( O(1) \) time (involving a comparison of the \((\text{age}, \text{rank})\) pair of the resource to that of the minimum valued \((\text{age}, \text{rank})\) in the specified bucket). Since the number of visits is bounded by \( \text{MAXQ} \) according to Theorem 5.2, if between one visit to another we would have checked in the
worst case $MAXQ$ buckets, then the total overhead would be $O(MAXQ^2)$. Next we justify our relaxation of the bucket list size. Specifically, we prove that checking a previously checked bucket is superfluous.

**Definition 5.9.** Let $schedule'$ be an algorithm identical to $schedule$ except to the following change: each time a bucket is visited or checked the $B$ set is left intact. Specifically, the following modifications are required to transform $schedule$ in Figure 5.7 to $schedule'$:

1. change the last bullet in 2(f) from $schedule(m, b_m, B - b_c)$ to $schedule(m, b_m, B)$.
2. remove 2(g).

**Theorem 5.5.** Let $b_k$ be a bucket to which one of the following applies in the course of calling $re-schedule$ with $schedule'$ as the auxiliary function rather than $schedule$:

1. $b_k$ is visited exactly once and checked afterwards at least once.
2. $b_k$ is checked more than once.

Then,

- if 1 applies then all checks are superfluous.
- if 2 applies then all checks except the first check are superfluous.

**Proof** We handle each of the above cases separately:

**case 1:** If $b_k$ is visited then by Theorem 5.2 it will not be visited again. Thus, by Definition 5.8, any future check of $b_k$ is superfluous.

**case 2:** Each check can be conceived to occur before a visit (the visit need not occur immediately after a check, that is, there could be other checks to other buckets in between). Let the first check of $b_k$ occur before visit in bucket $b_i$ and the second check occur before visit in bucket $b_j$. Note that $i \neq j$ by Theorem 5.2.
if $k = i$ : then $k$ is a bucket visited and then checked which is exactly case 1 and that completes the proof.

if $k \neq i$: we use the following notation:

- $r^i_{\text{in}}$ - the resource that is inserted to $b_i$ in the course of the visit there.
- $r^j_{\text{in}}$ - the resource that is inserted to $b_j$ in the course of the visit there.
- $rel_{r^i_{\text{in}}}$ - the relative $(age, rank)$ pair of $r^i_{\text{in}}$.
- $rel_{r^j_{\text{in}}}$ - the relative $(age, rank)$ pair of $r^j_{\text{in}}$.
- $r^m_k$ - the resource with the minimum $(age, rank)$ pair in bucket $b_k$.
- $rel_{r^m_k}$ - the relative $(age, rank)$ pair of $r^m_k$.

Since $r^i_{\text{in}}$ is checked with bucket $b_k$, but does not visit there, it must be that

$$rel_{r^m_k} \geq rel_{r^i_{\text{in}}}$$

Moreover, according to Corollary 5.4

$$rel_{r^i_{\text{in}}} > rel_{r^j_{\text{in}}}$$

Therefore, it follows that

$$rel_{r^m_k} > rel_{r^j_{\text{in}}}$$

Thus, the check of $b_k$ before $r^i_{\text{in}}$ visits $b_j$ is superfluous.
**Complexity considerations of re-schedule**

Let us consider the worst case scenario of **re-schedule**. Since each bucket is either visited or checked at most once, the worst case would be when each bucket is *visited* exactly once. When a bucket is visited, the resource with the minimum $(age, rank)$ in it is removed, another resource is inserted, and the removed resource is scheduled in another bucket. Suppose a bucket’s size is $n$. The removal of the minimum resource from the min-max heap thus costs $O(\log(n) + \log(S))$, since the update of $minR$, whose size is the number of servers $S$, is also necessary. Since $S << n$ we shall neglect $O(\log(S))$ and consider $O(\log(n))$ as the cost of removal. The insertion of another resource also costs $O(\log(n))$. Considering at most $MAXQ$ such operations, the overall cost of **re-schedule** is $O(MAXQ \cdot \log(n))$.

What is left is to estimate the value of $n$ in the worst case scenario. Since *NonUniform* distributes the resources evenly among buckets (last line in Figure 5.7), we conclude that the sum of resources *in all servers* in a single bucket is $\frac{S \cdot R}{MAXQ}$, i.e.

$$\forall i \in 0 \ldots MAXQ - 1 \sum_{s \in S} |RQ[i]| = \frac{S \cdot R}{MAXQ}$$

(5.1)

where $R$ is the number of resources per server and $S$ is either the set of servers or the size of it, according to the context. $|X|$ denotes the number of elements held in the data structure $X$. The worst case occurs if the summation in Equation 5.1 contains only a single server per bucket. However, a single bucket in a server can contain at most $R$ resources (since $R$ is the total number of resources per server), so we arrive at:

$$n = \min \left( \frac{S \cdot R}{MAXQ}, R \right)$$

Thus, the overall cost of **re-schedule** is
\[
MAXQ \cdot \log \left[ \min \left( \frac{S \cdot R}{MAXQ}, R \right) \right]
\]  \hspace{1cm} (5.2)

However, on average, the summation in Equation 5.1 will be distributed equally among the servers and thus the average complexity of \textit{re-schedule} is

\[
MAXQ \cdot \log \left( \frac{R}{MAXQ} \right)
\]  \hspace{1cm} (5.3)

\subsection*{5.1.4 Rank Calculation}

The \textit{rank} calculation is of prime importance in our algorithm. In Section 5.1.3, we saw the function \texttt{rank2nextCycle}, which calculates the next cycle to probe a resource based on its \textit{rank} field: the higher the \textit{rank}, the smaller the \textit{next-cycle} value, i.e. the higher the resulting probing frequency. \textit{Rank} also plays an important role in the \texttt{schedule} algorithm as we have seen in Section 5.1.3.

This Section deals with two aspects related to the Rank Calculation.

- \textbf{the \texttt{rank2nextCycle} function} - how should we set the extreme values for the \textit{next-cycle} corresponding to minimum rank and maximum rank values?

- \textbf{rank calculation} - on what should the rank of the resource depend?

\textbf{The \texttt{Rank2nextCycle} Function}

We have set forth the general characteristics of the \texttt{rank2nextCycle} function back in Section 5.1.3: as \textit{rank} increases from 0 to 100, \textit{next-cycle} should decrease from its maximum to minimum values. But what should those values be? The maximum possible value is bounded by \textit{MAXQ} – 1 (since we have at most \textit{MAXQ} buckets \textit{including} the current
bucket), and the minimum possible value is 1 (meaning the next cycle), but are these the best choices? Also, how should the function behave between these two extremes?

To gain some insight to this problem, let us further discuss the $MAXQ$ parameter, which is the size of our window. While there is no upper bound on $MAXQ$, other than system memory limitations (which of course are important in the practical sense), a lower bound can be obtained as follows:

The size of $MAXQ$ should be big enough to hold at least one probing and one processing of each resource, since each resource is scheduled at least once in the window (although the probing might not take place and a schedule can be missed). A unit probing and processing take $ts$ and $tp$ time, respectively, where $ts$ and $tp$ are our estimates of the request sending time and reply processing time. So, a single cycle with $C$ time units can support at most $C/(ts + tp)$ resources. Since the number of resources in the whole system is $S \cdot R$ (where $S$ is the number of servers and $R$ is the number of resources per each server), the total number of cycles needed in our window, and hence the lower bound for $MAXQ$, is set to $S \cdot R \cdot (ts + tp)/C$.

The quotient $S \cdot R \cdot (ts + tp)/C$ divided by $MAXQ$ will be called the system load.

As $MAXQ$ increases the load decreases and a load value of 1 denotes the situation where $MAXQ$ equals the minimum value of $MAXQ$ derived above.

Now it is obvious that $rank2NextCycle$ is closely related to the load of the system - if the system load is low (below 1) then a single window of size $MAXQ$ can hold for some resources, more than one probing and reply processing. For instance if the load is 0.5, the $MAXQ$ sized window can hold 2 probes and reply processing for each resource.

Figure 5.12 depicts the function we propose as the basis for experimentation for setting $rank2NextCycle$. 
Since \( rnk \) itself is calculated heuristically (discussed in the following section), we have chosen the function in 5.12 to be linear, following Occam’s razor principle (as mentioned in Section 5.1.3). \( MaxNCF \) and \( MinNCF \) are tunable parameters that are empirically evaluated in the simulations we conduct. Note that the maximum value of \( MaxNCF \) is \( \frac{1}{load} \).

**Rank Calculation**

In essence, \( NonUniform \) calculates the rank of a resource based on two factors, namely the existence probability and the resource fields, denoted by \( Ep \) and \( pop \).

We isolate these parameters and empirically seek to best set the rank as a function of these fields, in order to minimize the target function.

What \( NonUniform \) aims at is adapting the probing frequency to the need, where the need is obviously influenced by resource popularity since by definition, the target function is a popularity-weighted average of the deviations of estimated death time from true resource death time.
A more subtle factor that we find appropriate to introduce in the rank calculation formula is the existence probability parameter of the resource. Recall that each resource is associated with $E_p$, which gives a percentage estimate to the likelihood that the resource is alive.

We hypothesize that better results could be achieved by increasing the probing frequency for only part of the resources on selected time intervals, while lowering the probing frequency for other resources to compensate the increased system load. The resources gaining higher probing frequency should be those with lower values of $E_p$, i.e the resources that are close to their death, according to our estimation.

Chapter 6 starts with finding empirically the optimal function which incorporates both the $E_p$ and popularity of the resource, along with optimally configuring the tunable parameters in Figure 5.12.

### 5.2 The $E_{p\text{Combined}}$ Algorithm

The $E_{p\text{Combined}}$ algorithm estimates the $E_p$ parameter for each resource based on its a priori lifespan distribution estimates combined with the reply statistics of the servers hosting the resources. This is contrary to $E_{p\text{Dist}}$ (see Section 4.2), which is based upon the a priori lifespan distributions only.

Figure 5.13 describes the $E_{p\text{Combined}}$ scheme, which is applies after a resource is probed and when its reply is being processed. As before, we assume the mediator maintains a priori lifespan distributions of resources based on their type. The $E_p$ field of a resource $r \in R$ whose reply is being processed, is accessed by $r.E_p$.

We see that the major difference between $E_{p\text{Combined}}$ and $E_{p\text{Dist}}$ (see Figure 4.5) is in the handling of the ERROR reply status: while $E_{p\text{Dist}}$ uses in this case only the
- update the server reply statistics. (see Section 5.2.1)
- let status refer to the status of the reply, and $r$ be the subject resource
- check status:
  - case OK - leave $r.Ep$ unchanged
  - case DEAD - announce the resource as dead
  - case ERROR -
    * recalculate $r.Ep$ (see Section 5.2.2)
    * if $r.Ep < \text{threshold}$ then announce the resource as dead

Figure 5.13: $epCombined$ algorithm

a priori knowledge to set the new value of $Ep$, $EpCombined$ employs statistics of past probings.

The overall time complexity of $EpCombined$ is $O(1)$, since updating the reply server statistics (discussed in Section 5.2.1) and the $Ep$ calculation (discussed in Section 5.2.2) both take $O(1)$ time.

The following sections elaborate on the details: Section 5.2.1 explains what are the server reply statistics, and Section 5.2.2 describes the derivation of the $Ep$ re-calculation formula.

### 5.2.1 Server Reply Statistics Gathering

In Figure 5.13, the first action taken is the updating of the server reply statistics. Thus, these statistics are updated each time a reply is processed.

The mediator holds for each server in the system $s \in S$ its separate reply statistics information which is organized as follows: For each resource type, we maintain a counter for each type of status code received in past replies, i.e. counters for OK, DEAD, and
ERROR. This gives us the overall reply distribution per resource type in the server and is used in the calculation of $E_P$ in the following section.

### 5.2.2 $E_P$ Calculation Scheme

This section shows the derivation of the $E_P$ calculation scheme employed in the $E_{pCombined}$ algorithm (see Figure 5.13). We use the following notation:

- $P(\text{dead})$ - the probability of the resource being dead, i.e. permanently gone.
- $P(\text{alive})$ - the probability that the resource is alive. This is equivalent to $E_P (E_P = 100 \cdot P(\text{alive}))$.
  - Note that $P(\text{alive}) = 1 - P(\text{dead})$.
- $P(\text{Neg})$ - the probability of receiving a reply status code of either DEAD or ERROR from the server as a response to the head request for the resource.
- $P(\text{Pos})$ - the probability of receiving a reply status code of OK from the server as a response to the head request for the resource. $P(\text{Pos}) = 1 - P(\text{Neg})$.

We aim at calculating $P(\text{dead})$:

$$P(\text{dead}) = P(\text{dead}/\text{Neg})P(\text{Neg}) + P(\text{dead}/\text{Pos})P(\text{Pos}) \quad (5.4)$$

Now the second term in this sum is clearly zero since $P(\text{dead}/\text{Pos}) = 0$. So we are left with

$$P(\text{dead}) = P(\text{dead}/\text{Neg})P(\text{Neg}) \quad (5.5)$$
Note that if the server had no errors, $P(dead/Neg)$ would equal 1, and $P(dead)$ could simply be calculated by using the server reply statistics to estimate $P(Neg)$. In order to estimate $P(dead/Neg)$ in the presence of errors we factor out $P(dead/Neg)$ from the above equation and receive

$$P(dead/Neg) = P(dead)/P(Neg)$$

here $P(dead)$ is estimated by the resources’ type normal distribution estimate known a priori to the mediator, and

$$P(Neg) = P(Neg/dead) \times P(dead) + P(Neg/alive) \times P(alive) \quad (5.6)$$

$$= P(dead) + P(Neg/alive) \times (1 - P(dead)) \quad (5.7)$$

Since $P(Neg/dead) = 1$ and $P(alive) = 1 - P(dead)$

Finally, we arrive at

$$P(dead/Neg) = \frac{P(dead)}{P(dead) + P(Neg/alive) \times (1 - P(dead))} \quad (5.8)$$

Combining equations 5.5 and 5.8 we arrive at:

$$P(dead) = \frac{P(Neg)P(dead)}{P(dead) + P(Neg/alive) \times (1 - P(dead))} \quad (5.9)$$

Where we estimate $P(Neg/alive)$ by the server reply statistics by considering the statistics for the reply status code ERROR. Next, we show that indeed this estimate is legitimate.
\[ P(\text{Neg/alive}) = P(\text{ERROR} \cup \text{DEAD/alive}) \] (by definition of Neg)

\[ = \frac{P((\text{ERROR} \cup \text{DEAD}) \cap \text{alive})}{P(\text{alive})} \] (conditional probability definition)

\[ = \frac{P((\text{ERROR} \cap \text{alive}) \cup (\text{DEAD} \cap \text{alive}))}{P(\text{alive})} \] (distributive law)

\[ = \frac{P(\text{ERROR} \cap \text{alive})}{P(\text{alive})} \] \((\text{DEAD} \cap \text{alive}) = \emptyset\)

\[ = P(\text{ERROR/alive}) \quad \text{conditional probability definition} \]

\[ = P(\text{ERROR}) \quad \text{since ERROR and alive are independent events} \]

This concludes the derivation and we next give a qualitative explanation of this scheme.

Let us look at the formula for \( P(\text{dead}) \) given in Equation 5.9, as applied to a resource \( r \) of type \( \text{type} \) in server \( s \).

\[
P(t) = \frac{(p_{\text{ERROR}} + p_{\text{DEAD}}) \cdot p_{\text{norm}}}{p_{\text{norm}} + p_{\text{error}} \cdot (1 - p_{\text{norm}})}
\]

Where:

- \( P(t) \) - is \( P(\text{dead}) \) as a function of time.

- \( p_{\text{ERROR}} \) - this is the probability of receiving an \text{ERROR} reply for a resource of type \( \text{type} \) at server \( s \), as gathered by the server reply statistics, described in Section 5.2.1, at time \( t \) (note that all the server reply statistics are a function of time).

- \( p_{\text{DEAD}} \) - this is the probability of receiving a \text{DEAD} reply for a resource of type \( \text{type} \) at server \( s \), as gathered by the server reply statistics, at time \( t \).

- \( p_{\text{norm}} \) - this is the cumulative density normal distribution function at time \( t \) for resources of type \( \text{type} \), calculated based on the a priori resources lifespan according to their types.
To examine $P(t)$ let us look at a resource $r$ whose a priori normal distribution estimate is as follows:

- mean $= 10$ [time units]
- standard deviation $= 2.5$ [time units]

Figure 5.14 shows $p_{\text{norm}}$ as a function of time, with these parameters.

Figure 5.14: $p_{\text{norm}}(t)$ for mean=10, std=2.5

Suppose that our estimate is too early in the sense that the true normal distribution mean of the resources of type $r$ is larger than 10, say 15, and specifically, $r$ dies at time 14.

Consulting Figure 5.14 shows that for $t \geq 12$, $p_{\text{norm}}(t) \geq 0.8$, and eventually an $epDist$ algorithm with a threshold $\geq 0.2$ would kill the resource much too early leading to an $FN$ situation.

Now let us look at $P(t)$ for various time points as a function of $p_{\text{ERROR}}$ and $p_{\text{DEAD}}$ with the same mean and standard deviation of the type of $r$, $type$, as shown in Figure 5.15.

We can see that $P(t)$ increases both with $p_{\text{ERROR}}$ and $p_{\text{DEAD}}$, and that compared to Figure 5.14 the change as a function of time is milder. Specifically, examining time values
For $t \geq 12$, we can see that for values of $p_{DEAD} \leq 0.3$, $P$ is always smaller than 0.8, and for values of $p_{DEAD} \leq 0.4$ and $p_{ERROR} \leq 0.4$, again $P$ is always smaller than 0.8. It is reasonable to assume that in practice $p_{ERROR}$ will not be greater than $0.2 - 0.3$, in which case $P$ is smaller than 0.8 for all values of $p_{DEAD} \leq 0.5$. Now, considering that $type$ has mean 15, the probability of a resource to be dead until time 15 is 0.5. Thus, it is safe to assume that at most half of the resources will die until time 15, leading to $p_{DEAD} \leq 0.5$, and eventually that $P$, as calculated for resource $r$, would be smaller than 0.8. Therefore, the $epCombined$ algorithm with a threshold of 0.2 would not kill $r$ too early as did the $epDist$ algorithm. In this observation lies the superiority of the proposed algorithm over $epDist$. 

56
Figure 5.15: $P(t)$ for various times, as function of $p_{ERROR}$ and $p_{DEAD}$ [in %]
Chapter 6

Simulation

The purpose of this section is three-fold:

1. To show by simulation that algorithms belonging to \( W_p \) (recall the definition from Chapter 4) give results which are superior compared to algorithms in \( W_{np} \).

2. To find empirically the optimum rank calculation function for the \textit{NonUniform-EpCombined} algorithm, presented in Section 5.1.4.

3. To compare simulation results of the four algorithms achieved by combining the two validation schemes, namely \textit{Uniform} and \textit{NonUniform}, with the two \textit{Ep} calculation schemes, namely \textit{EpDist} and \textit{EpCombined}.

6.1 Simulation Set Up

We simulate a WAA environment according to the model described previously. The major system parameters, along with their description and values follows:

- \(#\textit{Servers} \) - the number of servers participating in the simulation, set to 25.
• \#Resources/Server - the number of resources in each server, set to 400, with a total of 10,000 resources in the simulation.

• threshold - the value of $E_p$ below which a resource is pronounced dead, set to 20%.

• death-toll - the percentage of dead resources from the total initial resource count required for halting the simulation, set to 70%.

• TCP-Conn, The size of the TCP-Connection pool available for the mediator, set to 25 (same as \#Servers).

• TCP-overhead, The time needed to establish a TCP Connection to a remote server, set to 350[msec].

• serverAvail, representing the server responsiveness (availability), is modeled according to a one-week behavior pattern that shows variations in responsiveness according to time of day, and also varies from weekdays to weekend, as presented in Figure 3.3. The values of serverAvail range from 0 to 1, meaning complete irresponsiveness to complete responsiveness, respectively.

• $T$, the computational cycle length, as described in Chapter 3, set to 120 [sec].

• $C$, the time allocated for probing-dedicated work in $T$, as described in Chapter 3, set to 2 [sec]. Figure 3.2, repeated in Figure 6.1 for ease of reference, describes the relationship between $C$ and $T$.

![Figure 6.1: the mediator’s working cycle](image)

59
• \(ts\), describes the estimated time required to send an HTTP head request asynchronously, set to 150 [msec].

• \(tp\), describes the estimated time required to process an HTTP reply from a server, set to 150 [msec].

• \(serverLat\), described in Chapter 3 and following [6], is a time-varying random variable. However, for the sake of simplicity we shall assume that the latency of each server is normally distributed with mean=10[sec] and std=2[sec], with no temporal variations.

• \(noise-to-mean-of-death\), whose description follows in Section 6.2, was set to 0.2.

• \(pop\), resource popularity is modeled by a Zipf distribution [4] with \(\alpha = 0.65\). We assume a constant popularity for each resource throughout its complete lifespan, ranging from 1 (minimum popularity) to 100 (maximum popularity).

We assume a set of 3 resource types, specifically HTML, IMAGE and SOUND. The lifespan of each type is normally distributed. HTML files have a smaller mean \(dt\) than images (50 days vs. 100 days according to [19]), and images have smaller mean \(dt\) than audio. The type of each resource in the system is one of the three above, and its \(dt\) is taken as a sample point from the relevant type normal distribution as shown is Figure 6.2. For the sake of simplicity, we consider the lifespan distributions to depend on resource types only (contrary to [19] which finds a dependence on resources’ domains as well).

As can be seen in Table 6.2, the nominal values for the lifespan distributions that we have chosen are smaller compared to those found in [19]. The motivation was simply to achieve empirical results faster. However, we have verified that similar results are achieved also with values found in [19], and thus concluded that the exact nominal values of the lifespan distributions’ mean is insignificant.
<table>
<thead>
<tr>
<th>resource type</th>
<th>normal distribution properties</th>
<th>mean</th>
<th>standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTML</td>
<td></td>
<td>10[days]</td>
<td>2[days]</td>
</tr>
<tr>
<td>IMAGE</td>
<td></td>
<td>15[days]</td>
<td>3[days]</td>
</tr>
<tr>
<td>SOUND</td>
<td></td>
<td>20[days]</td>
<td>4[days]</td>
</tr>
</tbody>
</table>

Figure 6.2: resources’ lifespan distributions

The algorithmic complexity overhead was taken into account by advancing the simulation time each time a manipulation on data structures occurs by an amount of time estimated with standard complexity consideration. To transform complexity counts to time, we use a constant multiplication factor $k$. For example, when extracting the minimum element of a minimum heap $H$, we advance the simulation time by $k \cdot \log(size)$, where $size$ is the size of $H$ at the time of extraction.

Note the algorithmic complexity is in addition to $ts$ and $tp$, which are taken into account by all validation algorithms, i.e. no matter the validation algorithm, each requesting sending costs at least $ts$ time, and each reply processing costs at least $tp$.

$k$ is calculated by estimating the cost of a single atomic operation in milliseconds. Considering a modern computer of a 2GHz processor speed, a single operation costs

$$\frac{1}{2GHz} = 5 \cdot 10^{-10} \text{[secs]} = 5 \cdot 10^{-7} \text{[msecs]}.$$ 

Knowing that the hidden factors in our complexity derivations are low, we can safely assume that a single operation would not cost more than 10 times than the above. Thus, we set $k = 5 \cdot 10^{-6} \text{[msecs]}$ in our simulations.

How much additional overhead lies in NonUniform compared to the Uniform validation scheme? In order to analyze the above we consider a large WAA system comprising of $S = 1000$ servers and a varying number of resources per server, ranging from
\( R = 25000 \) to \( R = 100000 \) resources, resulting in a total count of 25 million resources to 100 million resources overall.

The most expensive operation in NonUniform in either sending or processing a reply is \textbf{re-schedule} (Figure 5.6) which, on the average, according to Equation 5.3 costs

\[
O \left( MAXQ \cdot \log \left( \frac{R}{MAXQ} \right) \right)
\]

Figure 6.3: cost of \textbf{re-schedule}

Figure 6.3 shows the cost of such an operation, over a range of reasonable \( k \) values, for different values of \( R \). \( C \) was set to 60[msec], which is reasonable enough for such a large system, and the load was set to 0.5.

\( MAXQ \) was calculated with the formula:

\[
MAXQ = \frac{S \cdot R \cdot (ts + tp)}{C \cdot load}
\]
which was derived in Section 5.1.4. The time the operation costs was calculated by Equation 5.3, multiplied by the factor $k$.

We can see that even for $R = 100000$ (resulting in 100 million resources overall), and maximum value of $k$, the time needed for a single re-scheduling operation is not greater than 25[msecs], which is about 17% of the nominal $tp = 150$[msec]. Thus we conclude that the overhead induced by NonUniform is insignificant.

If a resource, $r$, is considered dead, then we do the following:

- record its death time $\hat{dt}$. This value will be compared to the actual death time of the resource $dt$ when the target function $F$ is calculated
- remove the resource from the list of resources of $s$, where $s$ is the server to which $r$ belongs
- spawn a new resource $\hat{r}$, and make it belong to $s$. This is done in order to preserve the load of the system

The system was simulated with JAVA JDK1.4 run under an AMD ATHLON ©XP 2800 Processor with 512MBytes of RAM memory and clock frequency of 2.1[GHz].

### 6.2 No Probing Results

Figure 6.4 shows the results achieved with an algorithm belonging to $W_{np}$, i.e. when no probing is used and $F$ is calculated based on a priori estimates of lifespan only. The results are given as a function of the noise-to-mean-of-death, i.e. the noise to the true normal distributions. The noise is expressed as a fraction of the mean of the normal distribution, where positive and negative noise refer to estimated mean that is higher and lower than the
true mean, respectively. For example, given a mean of 10[days], then a noise of 0.1 means that the estimated mean is 11[days]. If the noise is -0.2 than the estimated mean would be 8[days].

![Graph showing the relationship between noise and target function](image)

**Figure 6.4: no probing results**

As expected, as the noise to mean of death increases in its absolute value, the results deteriorate significantly. Compared to the results achieved later on with algorithms in $W_p$ (see Figure 6.14), we can deduce that algorithms in $W_{np}$ are not competitive with algorithms in $W_p$.

### 6.3 Rank Calculation Function Empirical Tuning

This section aims at finding and tuning the rank calculation function for the NonUniform-EpCombined algorithm, which is discussed in Section 5.1.4.
To gain a preliminary insight to the rank calculation problem we start with a rank calculation function that ignores both resource popularity and existence probability of the resources and simply schedules (and re-schedules) resources with equal cycle intervals. This is equivalent to simply using a constant rank value for each resource in Figure 5.12 (i.e. a horizontal line), and equivalent to a round robin scheme that probes all resources in a circular fashion.

What we do next is run the system with different load values obtained by adjusting the value of MAXQ, and in each run change the constant value of the next-cycle as in Figure 5.12. The purpose of this simple experiment is twofold:

- see the impact of the load of the system in the calculation of the rank-to-next cycle function.
- gain a base-line figure on which to improve the rank calculation function.

The results are summarized in Figure 6.5. We can clearly see that under constant next-cycle values the optimal scheduling value is always $1 \cdot \text{load} \cdot MAXQ$. Higher values of next-cycle, which result in lower probing frequencies achieve obviously poorer results, however smaller values of next-cycle equally achieve poor results since the system is overloaded and cannot handle the predetermined frequency. As a result, the system is more busy with the increased total operational overhead which can be seen in Figure 6.6.

Note that the maximum allowed value of next-cycle is always the window size MAXQ. Therefore in Figure 6.5 the x-axis values for each load are bounded from above by $\frac{1}{\text{load}}$. Consequently, only when the load $\leq 1$ is the optimal target function value of 27 (as can be observed in the figure) reached. Therefore, we conclude that under constant next-cycle function, results deteriorate as we increase load beyond 1.
The optimal value of 27 reached in the experiment can be explained as follows: The \textit{next-cycle} function returns a constant value of $load \cdot MAXQ$. In all the above experiments we used a constant value for the number of servers (25) and the number of resources in each server (400) rendering a total number of 10,000 resources in the system. The value of $MAXQ$ was set in such a way as to produce the required load. Hence, the \textit{next-cycle} function always produced, in the optimal case where $load \leq 1$, the number of cycles needed to probe and process the overall resources in the system, which can be expressed as $\frac{S \cdot R \cdot (ts + tp)}{C}$. Replacing the values for our setting, $S = 25$, $R = 400$, $ts = 150$ [msec], $tp = 150$ [msec], $C = 2$[sec] and $T = 120$[sec] we get 1500 cycles, which are equivalent to $1500 \cdot T = 50$ hours. Thus, each resource is probed every 50 hours. Since the true death time of a resource is equally likely to take place at any time point in the interval between 66
one probing to another, on the average it will take place in the middle of such an interval. Thus, if the resource is dead, its next probe would occur at most $50/2 = 25$ hours later on the average. The 27 hours we get in practice can be attributed to:

- **Server availability** - in our setting it was set to 90%. This means that some probes are lost in the sense that the reply status cannot indicate that the resource is dead.

- **The true death time distributions of the resources** - Recall that our model assumes that each resource type (e.g. sound file, image, etc.) has a typical lifespan which is normally distributed. As the standard deviation of these normal distributions decrease the target function converges to 25. A qualitative explanation to this behavior follows: suppose resources die exactly at the same time. Now after this death time the mediator starts probing all the resources in a round robin manner.
The estimated death time of the first resource probed will have zero deviation compared to the true death time (in case the server is always responsive - i.e. the server availability is 100%) since we assume the resource is probed instantly after it dies. The next resource will have a slight deviation, etc, until the last resource is probed. The latter will have a deviation equal to the time required to probe all resources, which in the above quantitative experiment is 50 hours. Hence the average will be halfway at 25 hours. Now suppose the resources do not die exactly at the same time but rather let the lifespan be normally distributed as in Figure 6.7. Also let \( x \) be some resource. Note that \( x \) is probed at time \( t_1 \) while it only dies at \( t_2 \) \((t_2 > t_1)\), hence the probe at \( t_1 \) will inform the resource is alive until the next probe, which will cause the higher deviation in the estimation of the resource’s lifespan.

![Figure 6.7: resources’ lifespan normally distributed](image)

68
6.3.1 Incorporating Resource Existence Probability $E_p$ into the Rank Calculation Function

In the previous section we saw that when a constant next-cycle value is used for all resources, the best results are achieved when $next-cycle = load \cdot MAXQ$. We also observed that decreasing next-cycle below the above value resulted in the deterioration of the target function $F$ which can be attributed to the increased load on the system. This stems from the fact that we have increased the probing frequency (due to the decrease of next-cycle) for all resources and with no time limits.

We follow the hypothesis presented in Section 5.1.4, which suggested to increase the probing frequency for resources we believe are going to die in the near cycles (i.e. when the value of $E_p$ decreases beyond a predetermined threshold). This increase will be accompanied by lowering the frequency below $load \cdot MAXQ$ for other resources since as we have seen in Figure 6.5, a probing frequency greater than $load \cdot MAXQ$ induces higher total operational overhead which results in worse target function values.

First we try a linearly decreasing rank function as in Figure 6.8, while the rank\_nextCycle parameters Figure 5.12 namely MaxNCF and MinNCF are chosen experimentally and compared. Parameter $t$ is the threshold $E_p$ value used in the algorithm. In order to isolate the influence of the $E_p$ factor, we set the popularity of all resources to a constant value.

The results are shown in Figure 6.9. We can see that as MaxNCF increases the results get worse. We also see that the MinNCF parameter does no seem to have a big impact on the results. The reason stems from the fact the rank linearly increases to a maximum of 100 as $E_p$ reaches the threshold value, but examining the $E_p$ changing pattern of resources reveals a slow decay, and hence this increase in rank, and the accompanying increase in probing frequency is not fully exploited. Thus, we offer a second $E_p$ to rank calculation function, as depicted in Figure 6.10. The tuner\_1 parameter determines the step $E_p$ value.
Next we simulate the system with varying values of the tuner parameter above, and also vary the MaxNCF and MinNCF of Figure 5.12.

The results are summarized in Figure 6.11. Regardless of MaxNCF, we can see that minimum $F$ values were achieved for a step point of 75. Also, the best results were achieved with MaxNCF and MinNCF values of 1.2 and 0.2, respectively.

We have demonstrated an improvement of 7 hours in $F$ by the introduction of the step function compared to the results achieved by the basic method and an improvement of 5 hours compared to the linear Ep function of Figure 6.8.

### 6.3.2 Incorporating the Resource Popularity into the Rank Calculation Function

In the previous section we have shown that changing the rank of the resource according to its Ep leads to better results. Recall that the resource popularity was maintained at a constant value for the sake of isolating the impact of Ep. In this section we show that incorporating the resource popularity pop into the rank calculation function can improve the algorithm behavior.

Recall that we modeled the popularity of the resources in our system to follow Zipf
We propose the function in Figure 6.12 as a multiplication factor for adapting the $E_p$ to $Rank$ function.

The configurable parameters in the Figure are:

- $stepPop$ - the popularity below which we decrease the rank of the resources by a factor of $minF$ ($minF < 1$).

- $minF$ - see explanation on $stepPop$ above.

In essence what this function does is maintaining the probing frequency for the more popular resources, while decreasing the probing frequency for the less popular resources. Thus, an improvement compared to the previous results is not expected unless $MinNCF$ is lowered. Whether this is worthwhile depends on whether the penalty received by decreasing the probing frequency for the less popular resources is smaller in factor compared to the improvement factor multiplied by the popularity ratio (popular to less popular), and thus is highly coupled with the resources’ popularity distribution.
As expected indeed the best results were achieved with a decreased value of $\text{MinNCF} = 0.1$, rather than 0.2. This increased probing frequency is compensated by the decreased probing frequency of the least popular resources.

The results achieved while varying the configurable parameters $\text{stepPop}$ and $\text{minF}$ with $\text{MaxNCF} = 1.2$ and $\text{MinNCF} = 0.1$ are shown in Figure 6.13. It can be seen that the best results are achieved when $\text{stepPop} = 15$ and $\text{minF} = 0.8$, in which case $F$ reduces to 17, which is an improvement compared to 20 achieved thus far. For lower and higher values of $\text{stepPop}$ we see a degradation of the results.

### 6.4 Analysis of Results

In this section we compare the simulation results for the various combinations of validation schemes and $E_p$ setting algorithms shown before. The performances are compared while tuning various parameters which are important in empirical evaluation of the algorithm.
6.4.1 Impact of Noise to Mean of Death

We start by investigating the robustness of the algorithms by comparing the algorithms’ sensitivity to the noise-to-mean-of-death parameter.

In Figure 6.14 we see that NonUniform-EpCombined, gives the best results in the whole range of noise: positive and negative as well. We can also see that EpDist algorithms converge to their EpCombined counterparts as the noise increases, while performing poorly in negative values of noise. This can be attributed to the fact that EpCombined...
slows down the speed of $Ep$ decay. So, when the noise is negative, the mediator thinks the resources should die earlier and indeed using only the normal distributions in determining $Ep$, as does $EpDist$, the resources are determined dead too early and this gives rise to the high (bad) $F$ values. However, when using $EpCombined$, this fast decay in $Ep$ is avoided and resources are not killed early.

We can see that under validation scheme $Uniform$ the same trend repeats when comparing $Uniform-EpCombined$ to $Uniform-EpDist$. Note that in any case the $NonUniform$ validation scheme outperforms the $Uniform$ validation scheme.

### 6.4.2 Impact of Resources’ Lifespan Distribution

In Figure 6.2 we have seen how the lifespan of the various resource types in the system is distributed. Note that we maintained the following invariant

$$\forall type \in \{HTML, IMAGE, SOUND\} \quad \frac{std[type]}{mean[type]} = 0.2$$
Next we alter this quotient and see the impact on the results. Figure 6.15 summarizes the results. Note that the noise to the mean of death is reset in this experiment to 0.2, as specified in Section 6.1.

We see that as the quotient between the standard deviation of the resources’ lifespan to their mean increases, the results obviously get worse. However, the increase rate is considerably more noted with the $EpDist$ algorithms. This can be explained by recalling that $EpDist$ algorithms set $Ep$ based solely on the a priori lifespan distributions of the resources and a wider distribution range increases the inaccuracy of their estimates. Note that when $std/mean$ increases beyond 1.2, $Uniform-EpCombined$ outperforms $NonUniform-EpDist$.

We have repeated the above experiment of various values of noise-to-mean-of-death and the same pattern repeated, but for negative value of noise-to-mean-of-death we noted that as $std/mean$ increased from 0.2 to 1, the results for $EpDist$ algorithms slightly im-
proved and from 1 to 2 they started deteriorating rapidly. This is contrary to the monotone deterioration for positive values of noise-to-mean-of-death. Figure 6.16 shows the results for noise-to-mean-of-death set to -0.5. This can be explained as follows: for negative values of noise-to-mean-of-death, EpDist concludes that resources die earlier than they truly do. However, as we increase the quotient std/mean more resources will die earlier. The worsening in the results is of course due to the fact that as std/mean increases more resources die also later, so there is a balance between the improving factors (more resources dying earlier) and the worsening factors (more resources dying later) when std/mean = 1. Below this value, the improving factors dominate, and above this value the worsening factors dominate.
6.4.3 Impact of Server Availability

Next we explore the impact of parameter serverAvail which embodies the overall chance to receive a reply of status ERROR and thus represents the server’s responsiveness. As previously mentioned, the default value we used for it in previous simulations was 0.9.

Next we simulate the system with serverAvail ranging from 0.1 to 1, and compare the results achieved with the various algorithms.

The results, depicted in Figure 6.17 show again that NonUniform-EpCombined gives the overall best results. We can see that the results for all algorithms improve as the serverAvail increases, since low values for the serverAvail act as a smoke screen from observing the true condition of the resource, i.e. whether it is still alive or not. Note
that as serverAvail decreases the chances of getting a reply of status ERROR increase, which increases our uncertainty regarding the true resource condition. Note that EpDist is also sensitive to server availability since as the server availability decreases, we get more ERROR replies at the expense of DEAD replies, and EpDist kills a resource when receiving a DEAD reply.

6.4.4 Impact of Number of TCP Connections

Recall that we focus on batched algorithms which aim at maximizing the utilization of the TCP connection pool available for the mediator. Figure 6.18 shows the results achieved when running the system with algorithms NonUniform-EpCombined and Uniform-EpCombined
Figure 6.17: simulation results comparison - varying server availability

(we have omitted the $EpDist$ counterparts because they show a similar pattern), while increasing the size of the TCP Connection Pool.

We see a stabilization in the results as the fraction $\frac{\#TCP-Conn}{\#Servers}$ reaches 1 since with a pool of connections equal in size to the total number of servers in the system, a connection can be maintained for each server, thus eliminating the need to re-create connections. We also note that as the TCP Connection Pool decreases below 1, $F$ does not seem to change at all for $Uniform-EpCombined$ while we see an increase in the value of $F$ for $NonUniform-EpCombined$. This stems from the fact that a $Uniform$ validation scheme is the optimal validation scheme as far as minimizing TCP connection overhead is concerned. To illustrate, consider there is only a single TCP connection available in the pool and let
the sequence of probes to perform be

\[ r_1^1, r_2^2 | r_3^1, r_4^2 | r_5^1, r_6^2 \]

where upper and lower indices denote the server and resource numbers, respectively, vertical lines separate between working cycles, and time constraints allow for a maximum of 2 probes per working cycle.

A possible Uniform validation scheme ordering of the sequence would be

\[ r_1^1, r_3^1 | r_5^1, r_2^2 | r_4^2, r_6^2 \]

since in Uniform once a server is visited, all resources residing on it are probed.
Thus, the number of TCP connections created (in case the pool contains only a single TCP connection) is 2. Note that we assume TCP connections are kept alive in the time intervals between consecutive cycles.

On the other hand the original sequence itself satisfies the requirements of a batched algorithm, since in each cycle the resources are trivially grouped according to their servers. Thus, the number of TCP connections created in this case would be 6 in the worst case (since the initial interleaving is one among many permutations that theoretically take place).

Nevertheless, NonUniform-EpCombined still performs better than Uniform-EpCombined.

6.4.5 Impact of Number of Resources Per Server

Recall that the default number of resources per server used in the simulation is 400. Figure 6.19 shows the results achieved when running the system with algorithms NonUniform-EpCombined and Uniform-EpCombined (we have omitted the EpDist counterparts because they show a similar pattern), while increasing the number of resource per server from 200 to 800.

We can see that both the Uniform and NonUniform validation schemes increase linearly in this range but the increase rate is notably slower in the NonUniform algorithm. This can be explained as follows: because Uniform visits each resource in the system in constant rate and we increase the total number of resources in the system linearly, the resulting probing rate of each resource also decreases linearly. However, the NonUniform validation scheme adapts the probing rate so that the maximum probing rate for each resource is around the time it is estimated to die, and since not all resources are estimated to die at the same time, NonUniform increases the rate to the resources in the "death time vicinity" at the expense of decreasing it for other resources at the same
Figure 6.19: simulation results comparison - varying number of resources per server

time. This decrease, however, is not crucial for the calculation of $F$ and thus the final result improves.
Chapter 7

Conclusions

We presented a model for a WAA environment and categorized the scope of all possible algorithms that conduct resources’ probing according to two main categories, namely the validation scheme and the $E_p$ calculation scheme. Each such category was further divided and we have shown how to construct various probing strategies with the combination of the schemes proposed in each category. We hypothesized that the NonUniform-$E_p$Combined algorithm - which adapts its probing frequency online (according to resource popularity and dynamic availability status changes) and in which the estimates of resource availability take into account both basic resource characteristics and the history of previous probings - would outperform the other proposed algorithms.

We have provided evidence supporting the correctness of our hypothesis by conducting detailed simulations which compared the performance of the various algorithms while altering diverse system parameters.

To the best of our knowledge, this is the first work that presented a WAA environment with stochastic analysis of HTTP page responses, incorporating network aspects such as network latencies and TCP connection overhead, and resources’ lifespan estimation sub-
ject to limited computational resources. We have shown that without period probing the quality of service delivered to the clients is poor, and proposed some means towards a significant improvement.

Future work includes handling a variety of system constraints such as a maximum number of probes allowed per resource, and computing resource *updates* patterns in a WAA environment with the aim of building estimates of update histories of all resources.
Bibliography


