MemGator - A Portable Concurrent Memento Aggregator

Cross-Platform CLI and Server Binaries in Go

Sawood Alam
Department of Computer Science
Old Dominion University
Norfolk, Virginia - 23529 (USA)
salam@cs.odu.edu

Michael L. Nelson
Department of Computer Science
Old Dominion University
Norfolk, Virginia - 23529 (USA)
mln@cs.odu.edu

ABSTRACT

The Memento protocol makes it easy to build a uniform lookup service to aggregate the holdings of web archives. However, there is a lack of tools to utilize this capability in archiving applications and research projects. We created MemGator, an open source, easy to use, portable, concurrent, cross-platform, and self-documented Memento aggregator CLI and server tool written in Go. MemGator implements all the basic features of a Memento aggregator (e.g., TimeMap and TimeGate) and gives the ability to customize various options including which archives are aggregated. It is being used heavily by tools and services such as Mink, WAIL, OldWeb.today, and archiving research projects and has proved to be reliable even in conditions of extreme load.

Keywords

MemGator; Memento; Aggregator; Web Archiving

1. INTRODUCTION

With the growth in the number of public web archives it is becoming important to provide a means to aggregate them for better coverage and completeness. The Memento protocol [3] provides a uniform API to lookup URIs in web archives. Due to the wide support of the Memento protocol in the archiving ecosystem, it is now easy to aggregate archives' holdings for any given query. However, current applications can either use an ad hoc aggregator implementation or rely on centralized services such as LANL's Time Travel portal[1] and ODU Memento Aggregator[2]. While centralized third party services are serving their purpose well, the convenience has the tradeoff of lack of customization and control such as the client application's inability to specify which archives are aggregated. Centralized services are usually good for general usage, but are not suitable for specialized purposes such as research or heavy traffic applications. For example, certain archives have IP-based traffic throttling policies that might limit the ability of centralized servers in case of heavy traffic. Similarly, the recent surge of OldWeb.today caused increased load on archives. As a result, one archive requested to be excluded from polling. This would have been an issue if they were using a centralized service.

```
$ memgator --format=JSON --verbose http://example.com/
{
  "original_uri": "http://example.com/",
  "self": "http://localhost:1208/timemap/json/http:...",
  "mementos": {
    "list": [
      {
        "datetime": "2002-01-20T14:25:10Z",
        "uri": "https://archive.is/20020120142510/ht..."
      },
---TRUNCATED---
$ memgator --arcs=./archives.json --log=./memgator.log \
>          --agent="MemGator:1.0 Test Run <@WebSciDL>" \
>          --host=localhost --port=1208 \
>          --contimeout=20s --restimeout=45s server
MemGator Server is listening at:
http://localhost:1208/timemap/{FORMAT}/{URI-R}
http://localhost:1208/timegate/{URI-R} [Accept-Datetime]
---TRUNCATED---
```

Figure 1: CLI and Server Mode Examples

There are a few open source aggregator implementations such as Memento Server[3] and Memento Java Client Library[4], but they are either outdated or require a server setup.

With these issues in mind, we created the MemGator tool that provides a standalone cross-platform binary without any external dependencies. It can be used as a one-off command to retrieve the response on the standard output or run as a web service to replicate necessary features of the centralized memento aggregator services (Figure 1). We tried to keep the service API as close to the LANL's Time Travel service as possible for greater interoperability. Both the modes (CLI and server) come with a handful of customization options that are documented in the binary itself and can be seen using standard help flag. One such configuration option is to supply a custom list of archives to be aggregated or use the archive profile [1] based archive ranking to query top-K archives only. We made the source code and binaries publicly available[5]. The tool is currently being used heavily in OldWeb.today and WAIL[6]. We are also running it as a web service[7] that is primarily being used by Mink [2].

[1]http://timetravel.mementoweb.org/guide/api/
[2]http://mementoproxy.cs.odu.edu/

[3]https://code.google.com/p/memento-server/
[4]https://github.com/ukwa/mementoweb-client-java
[5]https://github.com/oduwsdl/memgator
[6]http://machawk1.github.io/wail/
[7]http://memgator.cs.odu.edu/

JCDL '16 June 19-23, 2016, Newark, NJ, USA

ACM ISBN 978-1-4503-4229-2/16/06.
DOI: http://dx.doi.org/10.1145/2910896.2925452

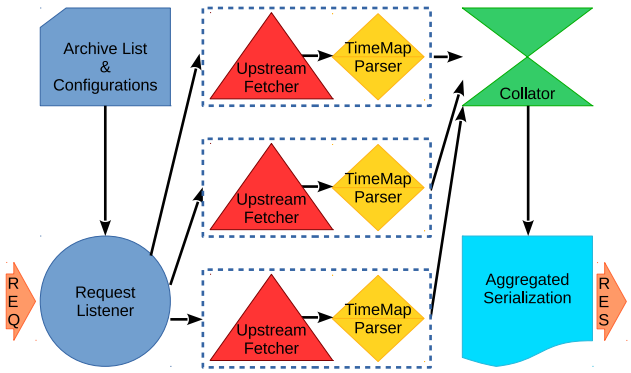

Figure 2: MemGator Workflow Diagram

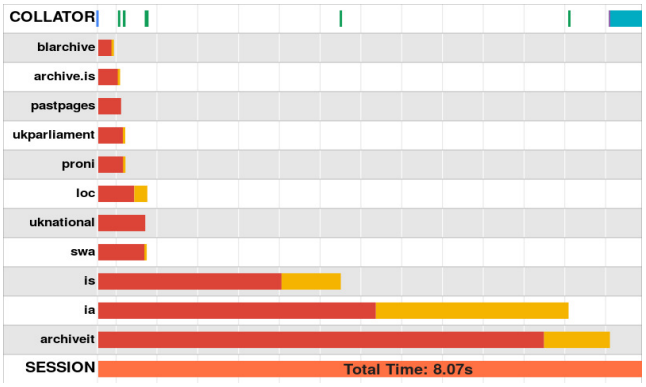

Figure 3: TimeMap Aggregation Request Timeline

2. IMPLEMENTATION

An aggregator is a good example of a concurrent application. It relies on various upstream archives which consumes the maximum amount of the overall time in network I/O while the process sits idle. Performing this operation sequentially will make it useless as the number of upstream services grows. We chose the Go language primarily because it is designed with concurrency in mind and has features that make development of concurrent web applications easy. Additionally, it provides the ability to create cross-compiled cross-platform static binaries.

Figure 2 illustrates the workflow of the MemGator implementation. The main thread (the request listener) loads the list of archives and other configuration options. When a lookup request is received, MemGator spins off goroutines (lightweight threads of Go) for each individual archive. These individual goroutines fetch the TimeMap from individual archives independently. If the response is successful the goroutine passes the data to a TimeMap parser via a channel (message passing mechanism of Go), which makes a linked list of the response in a chronological order. The parser sends the linked list data to the collator which accumulates responses from each individual goroutine and merges them while maintaining the sorting. Once all goroutines are completed or timeout occurs, the accumulator passes the aggregated linked list to the serializer. Depending on the format requested by the client (such as Link or JSON), the data is serialized and returned as the response to the user.

3. EVALUATION

We profiled individual functional blocks of a usual MemGator TimeMap request session with the microsecond precision and plotted them on a timeline to assess the gain of the concurrency. The top row of the Figure 3 shows activity in the main collator function when a response from an individual upstream goroutine is merged in the main linked list (while maintaining the canonical order). The far right activity in the first row is the time it took to serialize the response in the required format. The last row is the over all session time as observed by the MemGator. For a fairly large response (with 100,000+ Mementos) of `cnn.com` it took about 8 seconds. All the other middle rows show the time taken by the goroutines of individual archives for fetching the response (in red color) and parsing the fetched TimeMap (in yellow color) before passing the data to the main collator.

We then stress tested a server instance of MemGator using ApacheBench[8] for `cs.odu.edu` (with about 1,000 Mementos). Table 1 shows the number of requests MemGator was able to serve per second on various concurrency levels. Greater throughput on higher stress level is due to better utilization of the compute resources. For any individual request the processor is mostly sitting idle (and can be used for processing other requests), waiting for the network I/O to complete as illustrated in Figure 3 in red.

[8]https://httpd.apache.org/docs/2.2/programs/ab.html

Table 1: Stress Test Using ApacheBench

Concurrency	#Requests/sec (mean of 10 tests)
1	2.23
10	7.76
100	12.03
1000	64.70
>10000	ApacheBench I/O limit

4. FUTURE WORK AND CONCLUSIONS

The project repository has various feature requests that we need to assess and implement in a clean way while maintaining the interoperability with the existing tools to the extent possible. So far the MemGator implements all the basic features of a Memento aggregator (such as TimeMap and TimeGate) and gives the ability to customize various options including which archives are aggregated. It is being used heavily by tools and services such as Mink, WAIL, OldWeb.today, and archiving research projects and has proved to be reliable even in conditions of extreme load.

5. ACKNOWLEDGMENTS

This work is supported in part by the IIPC. Mat Kelly, Ilya Kreymer, Herbert Van de Sompel, and Harihar Shankar provided helpful feedback for the MemGator development.

6. REFERENCES

[1] S. Alam, M. L. Nelson, H. V. de Sompel, L. Balakireva, H. Shankar, and D. S. H. Rosenthal. Web Archive Profiling Through CDX Summarization. In *Proceedings of 19th International Conference on Theory and Practice of Digital Libraries, TPDL 2015*, pages 3–14.

[2] M. Kelly, M. L. Nelson, and M. C. Weigle. Mink: Integrating the Live and Archived Web Viewing Experience Using Web Browsers and Memento. In *Proceedings of the 14th ACM/IEEE-CS Joint Conference on Digital Libraries*, pages 469–470, 2014.

[3] H. Van de Sompel, M. L. Nelson, and R. Sanderson. HTTP Framework for Time-Based Access to Resource States – Memento. RFC 7089, Dec. 2013.