Benchmarking Grimoires

February 7, 2005

Abstract

In this document, the benchmarking methods, settings, and performance data of Grimoires are reported. The document is organized by benchmark methods.

1 Publication And Inquiry Overhead Against Registry Data Size

1.1 Method

Two basic operations on Grimoires registry are publication and inquiry. It is interesting to know the overhead of individual publication and inquiry operations with respect to the data size of the registry. We want to investigate to what extent the publication and inquiry overheads are affected when more and more data are registered into Grimoires.

In this test, we measure the overhead of publishing and querying service because we believe they are the most common operations in Grimoires' usage. In order to publish a UDDI service, three steps need to be performed:

1. Publish a UDDI business by invoking `save_business` defined in UDDI API specification,

2. Publish a UDDI TModel by invoking `save_tModel` defined in UDDI API specification, and

3. Publish a UDDI service by invoking `save_service` defined in UDDI API specification. The service belongs to the previously published business and refers to the previously published TModel.

Two steps are involved in querying a UDDI service:

1. Find a service by its name by invoking `find_service` defined in UDDI API specification, and

2. Retrieve the service detail by invoking `get_serviceDetail` defined in UDDI API specification. The input of `get_serviceDetail` is the service key returned by `find_service`.

Each step can be either a request and response pair of SOAP message in web service test, or a method invocation in business logic test.

In the test, we will repeat the following procedure:

1. Publish 100 services. Each published service has its unique name, description, binding template, TModel, etc.
2. Among all the services currently registered in Grimoires, randomly choose 100 services to query. To do this, we have maintain a name list for all the registered services. So we can find any service by the corresponding name.

1.2 Settings and performance data for business logic test

We use Jena 2.1 as the triple store back end. All the data are saved in memory. The maximal heap size of JVM is set to 1000MB by using the option of `java -Xmx1000m`. Axis 1.2Beta3 is used.

Figure 1 and 2 show the performance data for the test described in section 1.1. Figure 1 uses a linear y-axis scale, while figure 2 uses a logarithmic y-axis scale. We have the following observations from these figures:

Firstly, when there are 5000 services in Grimoires, it takes about 11.5 milliseconds to publish a service, and 434.4 milliseconds to query a service in business logic test; and it takes 61.9 milliseconds to publish a service, and 524.3 milliseconds to query a service in web service test. These figures give some idea how fast Grimoires is.

Secondly, compared with the publication overhead, the inquiry overhead is much more dependent on the registry data size. It is roughly proportional to the registry data size. This is due to Jena. The actual query will issue the following simplified RDQL statements:

```
SELECT ?businessService0 WHERE
    (?businessService0, <rdf:type>, <uddi:BusinessService>)
    (?businessService0, <uddi:hasName>, "some name")
```

To execute this query, Jena needs to go through all the RDF nodes which satisfy the statement that the node is of a type of `uddi:BusinessService`.

Thirdly, when the number of services in the registry is over 25000, the overhead suddenly becomes much bigger. It is caused by JVM garbage collection, as shown by the garbage collection curve. Garbage collection data are collected in business logic test. All the registry data are kept in the memory. Most of the heap might be allocated at that moment. So JVM has to do frequent garbage collections to allocate memory.

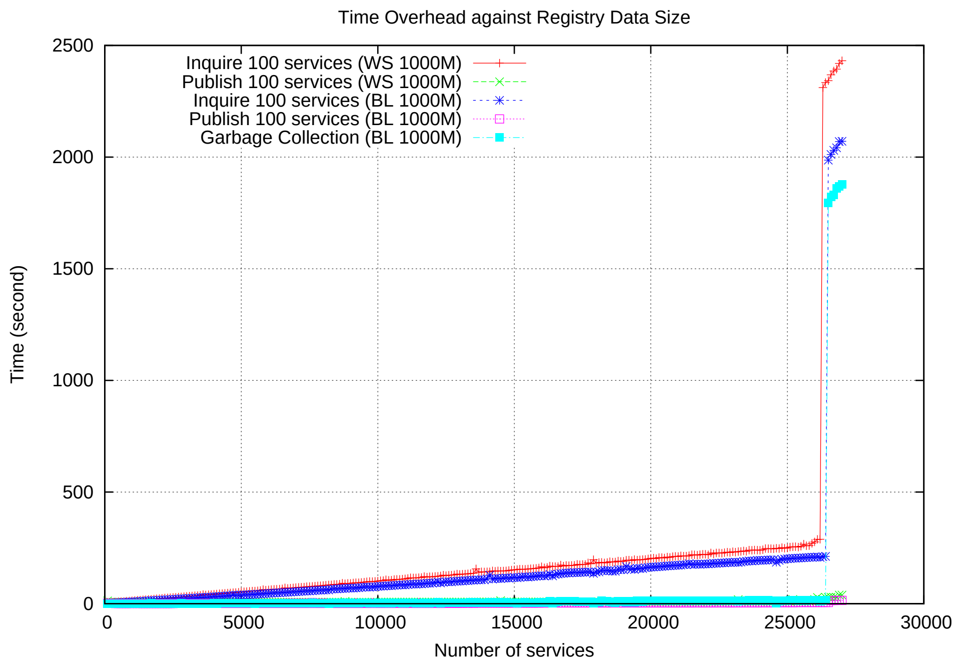

Figure 1: Time overhead with a linear y-axis scale

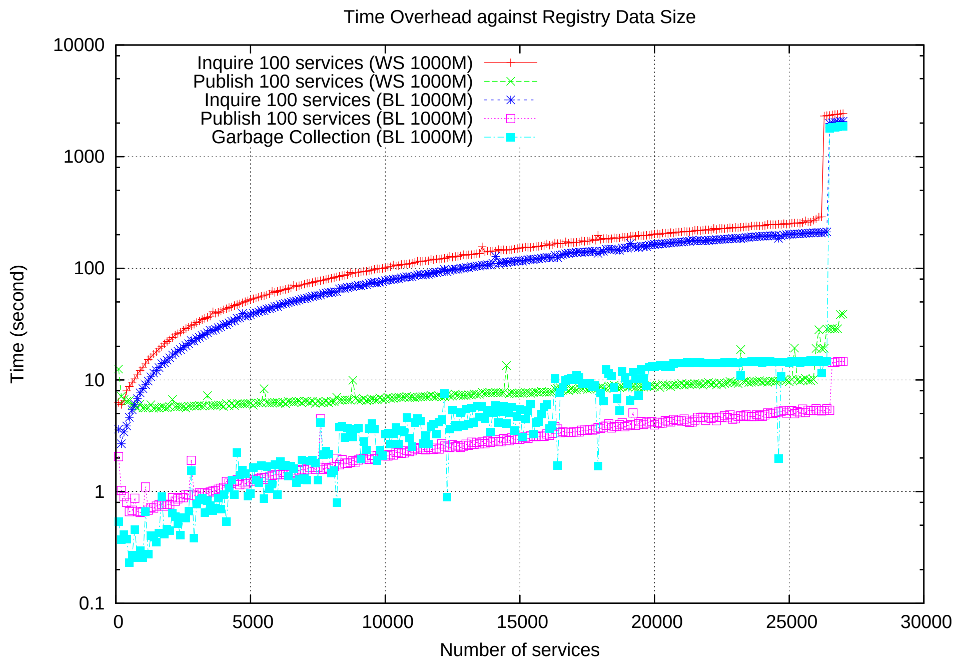

Figure 2: Time overhead with a logarithmic y-axis scale

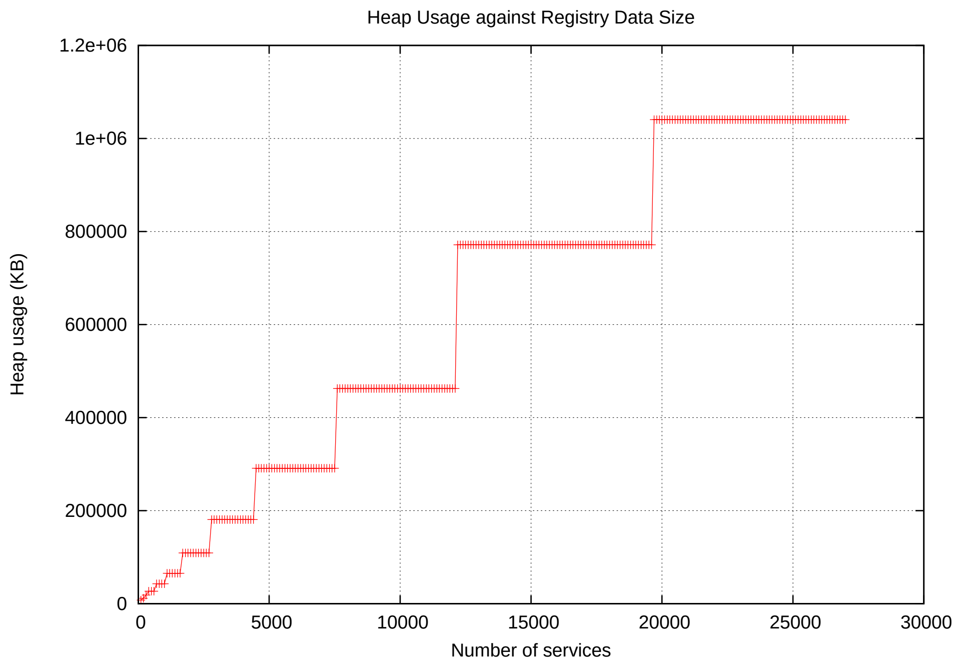

Figure 3: Heap usage

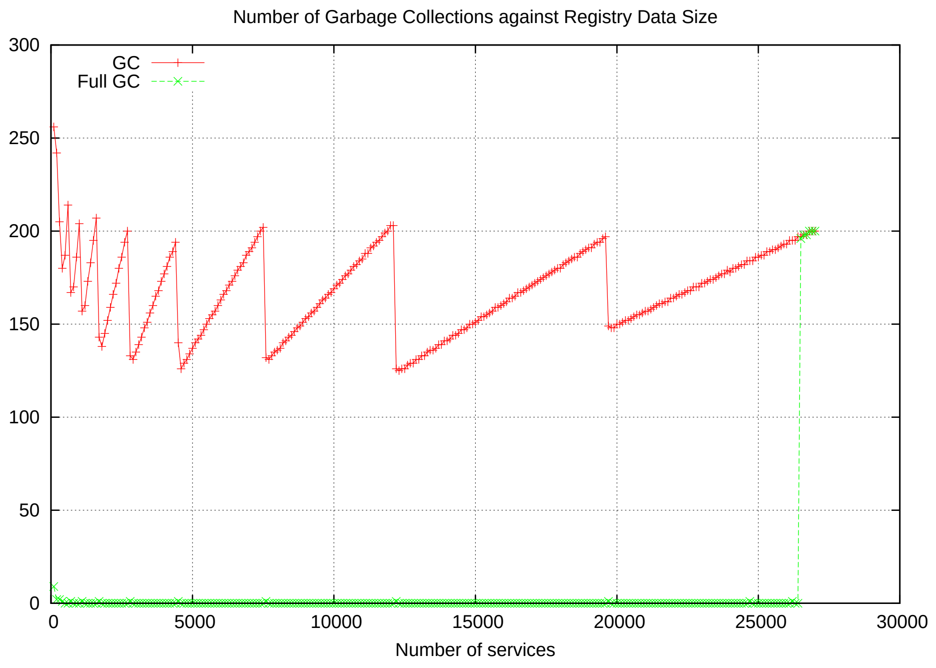

Figure 4: Number of garbage collections

Figure 3 shows the heap usage in business logic test. Figure 4 shows the number of grabage collections in business logic test.