Parliament User Guide

Ian Emmons

March 19, 2015

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Parliament is a high-performance triple store and reasoner designed for the Semantic Web.[1] Parliament[2] was originally developed under the name DAML DB[3] and was extended by BBN Technologies[4] for internal use in its R&D programs. BBN released Parliament as an open source project under the BSD license[5] on SemWebCentral[6] in 2009.

Parliament is a trademark of BBN Technologies, Inc., and is so-named because a group of owls is properly called a *parliament* of owls.

1.1 Background

The Semantic Web employs a different data model than a relational database. A relational database stores data in tables (rows and columns) while RDF[7] represents data as a directed graph of ordered triples of the form (subject, predicate, object). Accordingly, a Semantic Web data store is often called a graph store, knowledge base, or triple store.

[1] `http://www.w3.org/2001/sw/`
[2] `http://parliament.semwebcentral.org/`
[3] `http://www.daml.org/2001/09/damldb/`
[4] `http://bbn.com/`
[5] `http://opensource.org/licenses/bsd-license.php`
[6] `http://semwebcentral.org/`
[7] `http://www.w3.org/RDF/`

A relational database can store a directed graph, and some graph stores are in fact implemented as a thin interface layer wrapping a relational database. However, the query performance of such implementations is usually poor. This is because the only straightforward way to store the graph with the required level of generality is to use a single table to store all the triples, and this schema tends to defeat relational query optimizers.

Early in the Semantic Web's evolution, BBN encountered exactly this problem, and so the graph store we now call *Parliament* was born. The goal of Parliament was to create a storage mechanism optimized specifically to the needs of the Semantic Web, and the result was a dramatic speed boost for BBN's Semantic Web programs. Since its initial conception, Parliament has served as a core component of several projects at BBN for a number of U.S. Government customers.

1.2 Storage and Inference

Parliament implements a high-performance storage engine that is compatible with the RDF and OWL[8] standards. However, it is not a complete data management system. Parliament is typically paired with a query processor, such as Sesame[9] or Jena,[10] to implement a complete data management solution that complies with the RDF, OWL, and SPARQL[11] standards for data representation, ontology, and query, respectively.

In addition, Parliament includes a high-performance rule engine, which applies a set of inference rules to the directed graph of data in order to derive new facts. This enables Parliament to automatically and transparently infer additional facts and relationships in the data to enrich query results. Parliament's rule engine currently implements all of RDFS inference plus selected elements of OWL-Lite.

[8] http://www.w3.org/2007/OWL/

[9] http://www.openrdf.org/

[10] http://openjena.org/

[11] http://www.w3.org/2009/sparql/wiki/Main_Page

1.3 Parliament Architecture

Figure 1.1 depicts the layered architecture of Parliament. Parliament is a triple store and a rule engine, but it does not include a query processor. Therefore, it is typically paired with a third-party query processor, such as Jena or Sesame. The core of Parliament is written in C++, but the integration layers for Jena and Sesame are Java code. The Jena integration includes some useful extras, such as support for named graphs and temporal, geospatial, and numerical indexes.

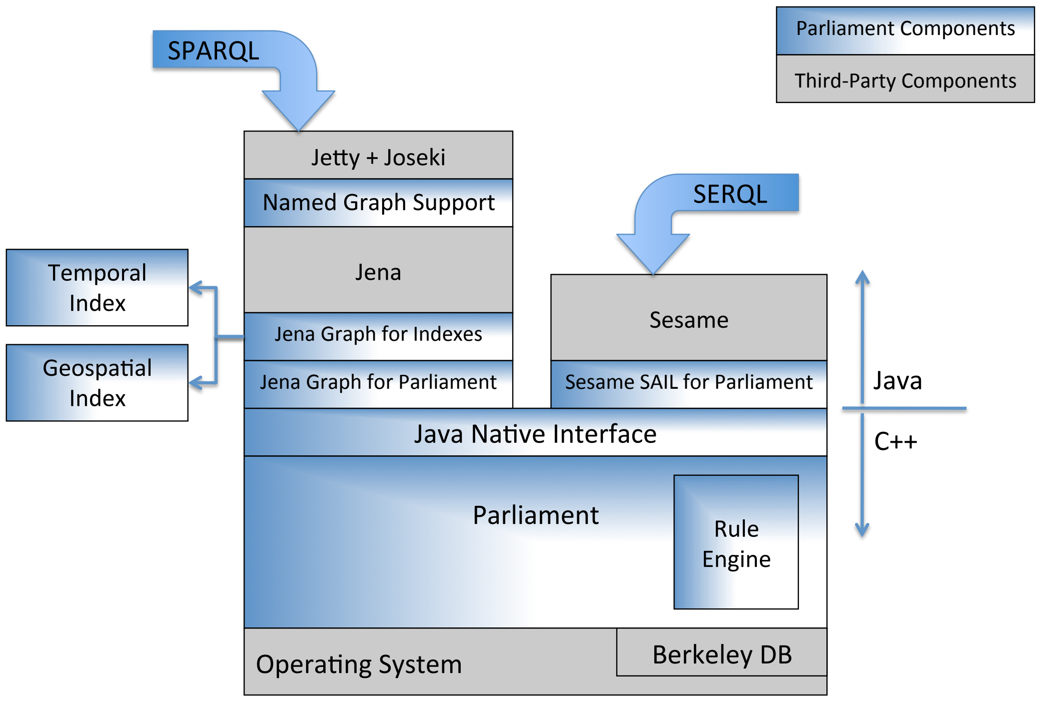

Figure 1.1: Layered Parliament Architecture

Chapter 2

Deploying and Using Parliament

There are several ways to use Parliament. By far the most common (and the quickest) is to download a Parliament "quick start" binary package. This combines Parliament binaries with Jena (a SPARQL query processor) and Jetty (a servlet engine) to create a complete knowledge base server application that implements a SPARQL endpoint. See Section 2.1 for complete instructions.

Parliament can also be used with other servlet engines (e.g., Tomcat and Glassfish — see Section 2.4.1) and with the Sesame query processor (see Section 2.4.2).

Yet another way to use Parliament is as an in-process library, again paired with a query processor. Less frequently, Parliament is used as an in-process library by itself, without the benefit of a query processor. See Section 2.5 for a discussion of these topics.

This chapter assumes that you have a binary Parliament distribution, and shows you how to deploy and use Parliament from it. There are two kinds of binary distributions:

- The "quick start" distribution, for easy deployment of a knowledge base server and SPARQL endpoint
- The complete distribution, containing everything necessary for any deployment scenario

Also note that there are two ways to acquire a binary distribution:

- Download pre-built binaries from the Parliament web site[1]
- Build them yourself (see Chapter 3)

The former is recommended for Windows and MacOS installations. The latter is generally the most appropriate choice for Linux.

This chapter starts with instructions for installing a "quick start" Parliament server (Section 2.1). Following that is a discussion of the Parliament configuration file (Section 2.2). Then comes a detailed discussion of each of the various deployment scenarios. The chapter concludes with discussions of Parliament utilities and troubleshooting (Sections 2.7 and 2.8).

Note that as of this writing, Parliament supports Jena 2.6 and Sesame 1.2. In particular, Sesame 2.0 and later are not yet supported.

2.1 Parliament Server Quick Start

2.1.1 Installation

A "quick start" distribution of Parliament is a compressed archive containing a ready-to-run Parliament Server installation, with the following naming convention:

`ParliamentQuickStart-v`***`X.Y.Z-toolset.`***`zip`

Here ***`X.Y.Z`*** is the Parliament version number and ***`toolset`*** indicates the platform on which the distribution runs, as shown in Table 2.1. If you are building Parliament yourself according to the instructions in Chapter 3, then these archives appear in the `target/distro` directory at the end of the build process.

As indicated in Table 2.1, most platforms require that you choose between 32-bit and 64-bit builds. Whenever possible, use the 64-bit version, because 32-bit Parliament installations are limited to 5 to 10 million statements.

Once you have the appropriate "quick start" distribution, the following steps will yield a functioning server:

[1]`http://parliament.semwebcentral.org/`

Toolset	*Corresponding Platform*
`msvc12-64`	64-bit Windows, built with Visual Studio 2013
`msvc12-32`	32-bit Windows, built with Visual Studio 2013
`clang`	Universal binaries for MacOS 10.8 (Mountain Lion) and later
`darwin`	Universal binaries for MacOS 10.7 (Lion) and prior
`gcc-64`	64-bit Ubuntu Linux
`gcc-32`	32-bit Ubuntu Linux

Table 2.1: Toolset-to-Platform Correspondence

1. Extract the archive contents to whatever location you choose. On Windows, `C:\Program Files\ParliamentKB` is a likely choice. On UNIX-like systems, `/usr/local/ParliamentKB` is customary. The instructions that follow refer to this directory as `ParliamentKB`.
2. On Windows, install the C and C++ run-time libraries. You can find installers for these in the following directory:

   ```
   ParliamentKB/RedistributablePackages
   ```
3. Invoke the startup script `StartParliament.sh` (on Windows, this is called `StartParliament.bat`), located in the `ParliamentKB` directory, to start the Parliament server.[2]

2.1.2 Usage

Please note: When you want to shut down the KB, it is important to shut it down gracefully. Otherwise, the Parliament files will not be flushed to disk before they are closed, and they will almost certainly be badly corrupted. If you run Parliament as a Windows service or UNIX Daemon (see Section 2.1.3), this is generally not a problem, because the operating system sends a shut-down message at the appropriate times. If, however, you run Parliament explicitly via `StartParliament.sh`

[2]On all platforms, these scripts locate your Java installation by simply invoking the command `java` and relying on the operating system's native facilities for locating the JVM executable. See Section 2.8 for more information.

(or `StartParliament.bat` on Windows), you will need to shut it down yourself by typing the command “exit” followed by «return» or «enter».

To use the Parliament server, you will need the the connection URLs listed in Table 2.2. (See Section 2.1.3 to configure your server to run on a host name and port other than “localhost” and “8080”.) You can use the Parliament server interactively

HTTP interface:	`http://localhost:8080/parliament`
SPARQL endpoint:	`http://localhost:8080/parliament/sparql`
Bulk loader:	`http://localhost:8080/parliament/bulk`

Table 2.2: Joseki Server Connection URL’s

by pointing your web browser at the HTTP interface URL from Table 2.2. This simple and (hopefully) self-explanatory web interface is useful for initial data loading, experimenting with queries, exploring a collection of data, troubleshooting, and similar tasks.

In order to issue queries programmatically, you will need to write some code to send SPARQL-compliant HTTP requests. One library that can send such requests on your behalf is Jena itself. Figure 2.1 shows sample code for issuing a SPARQL select query. Note that this code uses the SPARQL endpoint URL from Table 2.2.

```
import com.hp.hpl.jena.query.QueryExecution;
import com.hp.hpl.jena.query.QueryExecutionFactory;
import com.hp.hpl.jena.query.ResultSet;

void issueSelectQuery(String sparqlSelectQuery)
{
  QueryExecution exec = QueryExecutionFactory.sparqlService(
    "http://localhost:8080/parliament/sparql",
    sparqlSelectQuery);
  for (ResultSet rs = exec.execSelect(); rs.hasNext(); rs.next())
  {
    // Do something useful with the result set here
  }
}
```

Figure 2.1: Issuing a SPARQL select query with Jena

Parliament also provides a Java library to help programmers interact with the Parliament server, which can be found here:

```
ParliamentKB/lib/JosekiParliamentClient.jar
```

This library leverages the Jena functionality shown above, but exposes a few extra features of the Parliament server that are not accessible within the bounds of SPARQL. To use this approach, first add the above jar file to your class path. Then create a `RemoteModel` object as shown in Figure 2.2. This object allows you to access and manipulate the Parliament repository identified by the URLs passed to the constructor, which are again taken from Table 2.2. Note that the query

```
import com.bbn.parliament.jena.joseki.client.RemoteModel;

void useRemoteParliamentModel()
{
  // Create model:
  RemoteModel rmParliament = new RemoteModel(
    "http://localhost:8080/parliament/sparql",
    "http://localhost:8080/parliament/bulk");

  // Insert contents of another Jena Model:
  rmParliament.insertStatements(myPopulatedJenaModel);

  // Perform SPARQL SELECT query:
  ResultSet results = rmParliament.selectQuery(myQueryString);

  // Add statements using SPARQL-UPDATE:
  rmParliament.updateQuery(mySparqlUpdateQueryString);
}
```

Figure 2.2: Using Parliament Via a Remote Jena Model

strings passed to the remote model in Figure 2.2 follow standard SPARQL and SPARQL-UPDATE formats, including support for named graphs. Also note that the `RemoteModel` class exposes the bulk loading feature of the Parliament server, allowing the loading of large bodies of RDF without running afoul of the limitations of Joseki's SPARQL UPDATE implementation.

2.1.3 Configuration

This section explores the most important configuration options for your Parliament server. The "quick start" installation procedure in Section 2.1.1 simply accepts the defaults for these options, but most deployments will require some level of customization to be useful.

Section 2.1.1 mentions Parliament's start script, which is used to directly start the server. However, in many cases it is preferable to run Parliament as a Windows service or as a UNIX daemon.

The script `InstallParliamentService.bat` installs Parliament as a Windows service. (In order for this script to have the permissions it requires, you need to right-click the Command Prompt shortcut and choose "Run as Administrator.") This script locates your Java installation from the `JAVA_HOME` environment variable.[3] After running this script, Parliament can be started from the Services management console. You can also use this console to configure Parliament to run automatically upon system startup. As you might expect, you can uninstall the service with the script `UninstallParliamentService.bat`.

Yet to be written: How to run Parliament as a UNIX daemon.

You can change the host name (or IP address) and port of the SPARQL endpoint by setting `JETTY_HOST` and `JETTY_PORT` in the start script. These default to `localhost` and 8080, respectively. Also, you can control the amount of memory set aside for the Java virtual machine by setting `MIN_MEM` and `MAX_MEM` in the same location. While it is important to allow the JVM sufficient memory, it is also important to realize that a significant portion of Parliament is native code, and therefore does not use the Java heap. Java programmers are often inclined to set `MAX_MEM` close to the total memory of the machine, but this will starve Parliament's native code of the memory it needs to achieve good performance. These settings default to 128 MB and 512 MB, which are reasonable values for machines with 4 to 8 GB of total memory.

Note that if you are running the server as a Windows service, you must change these parameters in the install script, and you will have to uninstall the service and reinstall it to make your changes effective.

Yet to be written: How to change these parameters for Parliament as a UNIX daemon.

The file `ParliamentKB/conf/jetty.xml` holds the Jetty servlet container's configuration. Generally, there is no need to modify this file. Finally, the Parliament configuration file (`ParliamentConfig.txt`) is discussed in detail in Section 2.2.

[3] See Section 2.8 for more information about choosing among multiple Java installations.

2.2 The Parliament Configuration File

Like many pieces of software, Parliament has a configuration file. It is typically called `ParliamentConfig.txt`, but it can be named anything. Here is how Parliament finds its configuration:

1. If the environment variable `PARLIAMENT_CONFIG_PATH` is set, then its value is assumed to be the absolute path of the configuration file.
2. *On Windows only:* If this environment variable is not set, then Parliament looks for the file `ParliamentConfig.txt` in the same directory as the Parliament DLL.
3. If none of the above apply, then Parliament looks in the current working directory of the server process for the file `ParliamentConfig.txt`.

The configuration file is a plain text file containing name-value pairs, comments (preceded by a '#'), and blank lines. Many of the settings are Boolean values, in which case the value is case-insensitive and may be "true", "t", "yes", "y", "on", or "1" (for true), or "false", "f", "no", "n", "off", or "0" (for false).

The recognized settings are the following. (Note that the setting names are case-insensitive.)

kbDirectoryPath The path of the Parliament knowledge base files. (If they do not yet exist, they will be created here.) Note that this directory must exist before Parliament is started, or an exception will be thrown. *Default: The current working directory, "."*

stmtFileName The name of the memory-mapped file containing records of statements. *Default:* `statements.mem`

rsrcFileName The name of the memory-mapped file containing resource records. *Default:* `resources.mem`

uriTableFileName The name of the memory-mapped file containing resource strings (URIs and literals). *Default:* `uris.mem`

uriToIntFileName The name of the file containing the mapping from numeric resource identifiers to resource strings. This file is managed by Berkeley DB. *Default:* `u2i.db`

stmtToIdFileName The name of the file containing the mapping from tuples of numeric resource identifiers to numeric statement identifiers. This file is omitted by default, and is included only if the “keepDupStmtIdx” setting is turned on. This file is managed by Berkeley DB. *Default:* `stmt2id.db`

readOnly When set to “yes”, prevents Parliament from changing the underlying storage files in any way. *Default: “no”*

fileSyncTimerDelay The number of milliseconds between flushings of the KB files to disk. The flush is performed asynchronously, and so has minimal impact on overall performance. This decreases the chances of a file corruption, and it limits the amount of time required to shut down Parliament gracefully. Set to zero to disable flushing the files to disk. This setting applies only to Parliament deployed as a server using Jena, Joseki, and Jetty. Any other deployment will ignore this setting. *Default: “15000”*

keepDupStmtIdx When set to “yes”, Parliament will maintain an extra file which maps tuples of resources to statement identifiers. This allows Parliament to quickly decide if a statement assertion is referencing an already-asserted statement, or if a new statement must be inserted. Generally, this does not result in a noticeable speedup, because the default method for checking for an existing statement (by traversing the shortest of the subject, predicate, and object linked lists) is plenty fast enough. This option also results in a dramatic increase in the amount of disk space consumed. *Default: “no”*

initialRsrcCapacity The number of resources Parliament should allocate space for when creating a new KB. *Default: “100000”*

avgRsrcLen The average resource length (in characters) that Parliament should anticipate when allocating space in a new KB. *Default: “64”*

rsrcGrowthFactor The factor by which to increase the resource table size when Parliament runs out of space in the file. This must be larger than one. *Default: “1.25”*

initialStmtCapacity The number of statements Parliament should allocate space for when creating a new KB. *Default: “500000”*

stmtGrowthFactor The factor by which to increase the statement table size when Parliament runs out of space in the file. This must be larger than one. *Default: “1.25”*

bdbCacheSize The amount of memory to be devoted to the Berkeley DB cache. The portion before the comma is the total cache size (with a k for kilobytes, m for megabytes, g for gigabytes). The portion after the comma specifies how many segments the memory should be broken across, for compatibility with systems that limit the size of single memory allocations. On systems with 4GB or more of memory, setting this to "256m,1" generally seems to be optimal. *Default: "32m,1"*

runAllRulesAtStartup Causes Parliament to evaluate each of the enabled rules at startup against the existing statements in the KB to see if any new statements should be materialized. Generally, this is unnecessary, because the rules are always evaluated against each statement as it is asserted. However, if some of the rules were initially disabled when statements were being asserted, and are then enabled, it may be necessary to turn this on for one startup of Parliament to ensure that the state of the KB is in sync with the rule settings. Alternately, in such a case this setting may be left off and the ParliamentAdmin tool may be run against the KB with the "guaranteeEntailments" option. Note that when this setting is turned on, KB the startup time may be lengthy. *Default: "no"*

SubclassRule Enables the following inference rules:

$$A \subset B \land B \subset C \Rightarrow A \subset C$$

$$X \in A \land A \subset B \Rightarrow X \in B$$

where $\subset$ means "is a sub-class of" and $\in$ means "is of type". *Default: "on"*

inferRdfsClass When both this setting and "SubclassRule" are turned on, this enables the following inference rule:

$$A \subset B \Rightarrow A \in \text{rdfs:Class} \land B \in \text{rdfs:Class}$$

where $\subset$ means "is a sub-class of" and $\in$ means "is of type". *Default: "off"*

inferOwlClass When both this setting and "SubclassRule" are turned on, this enables the following inference rule:

$$A \subset B \Rightarrow A \in \text{owl:Class} \land B \in \text{owl:Class}$$

where $\subset$ means "is a sub-class of" and $\in$ means "is of type". *Default: "off"*

inferRdfsResource When both this setting and “SubclassRule” are turned on, this enables the following inference rule:

$$A \subset B \Rightarrow A \subset \text{rdfs:Resource} \wedge B \subset \text{rdfs:Resource}$$

where $\subset$ means “is a sub-class of”. *Default: “off”*

inferOwlThing When both this setting and “SubclassRule” are turned on, this enables the following inference rule:

$$A \subset B \Rightarrow A \subset \text{owl:Thing} \wedge B \subset \text{owl:Thing}$$

where $\subset$ means “is a sub-class of”. *Default: “off”*

SubpropertyRule Enables the following inference rules:

$$P \sqsubset Q \wedge Q \sqsubset R \Rightarrow P \sqsubset R$$

$$P \sqsubset Q \wedge P(X, Y) \Rightarrow Q(X, Y)$$

where $\sqsubset$ means “is a sub-property of”. *Default: “on”*

DomainRule Enables the following inference rule:

$$\text{rdfs:domain}(P, C) \wedge P(X, Y) \Rightarrow X \in C$$

Default: “on”

RangeRule Enables the following inference rule:

$$\text{rdfs:range}(P, C) \wedge P(X, Y) \Rightarrow Y \in C$$

Default: “on”

EquivalentClassRule Enables the following inference rule:

$$\text{owl:equivalentClass}(A, B) \Rightarrow A \subset B \wedge B \subset A$$

where $\subset$ means “is a sub-class of”. *Default: “on”*

EquivalentPropRule Enables the following inference rule:

$$\text{owl:equivalentProperty}(P, Q) \Rightarrow P \sqsubset Q \wedge Q \sqsubset P$$

where $\sqsubset$ means “is a sub-property of”. *Default: “on”*

InverseOfRule Enables the following inference rule:

$$\text{owl:inverseOf}(P, Q) \wedge P(X, Y) \Rightarrow Q(Y, X)$$

Default: "on"

SymmetricPropRule Enables the following inference rule:

$$P \in \text{owl:SymmetricProperty} \wedge P(X, Y) \Rightarrow P(Y, X)$$

Default: "on"

FunctionalPropRule Enables the following inference rule:

$$P \in \text{owl:FP} \wedge P(Z, X) \wedge P(Z, Y) \Rightarrow \text{owl:sameAs}(X, Y)$$

where "FP" stands for "FunctionalProperty". *Default: "on"*

InvFunctionalPropRule Enables the following inference rule:

$$P \in \text{owl:IFP} \wedge P(X, Z) \wedge P(Y, Z) \Rightarrow \text{owl:sameAs}(X, Y)$$

where "IFP" stands for "InverseFunctionalProperty". *Default: "on"*

TransitivePropRule Enables the following inference rule:

$$P \in \text{owl:TransitiveProperty} \wedge P(X, Y) \wedge P(Y, Z) \Rightarrow P(X, Z)$$

Default: "on"

2.3 General Binary Distribution Contents

Section 2.1 discusses the most common Parliament deployment scenario and its specialized binary distribution, but there are many other scenarios introduced in the sections below. The general binary distribution, introduced here, supports these scenarios.

The general distribution of Parliament is a compressed archive containing a number of different artifacts. (If you are building Parliament yourself according to the instructions in Chapter 3, then the contents of the `target/artifacts` directory are identical to the contents of a binary distribution.) After extracting the archive contents, you will see the following:

- This document (ParliamentUserGuide.pdf)
- One or more directories of native build products: Each such directory contains the native build products for one compiler (libraries, shared libraries, executables, and the like). The directory names correspond to the designation for the compiler according to Boost.Build. For instance, on Windows you will see `msvc-12.0` (Visual Studio 2013). On Macintosh, you will see `clang`. On Linux, you will see `gcc`.
- An `include` directory: This contains the header files required to write C++ code that directly calls Parliament. These are rarely used, because Parliament is usually accessed by Java code through its JNI interface.
- On Windows, a `RedistributablePackages` directory: This directory contains installers that update the run-time libraries on which the native build products depend. See Section 2.8 for details.
- Jar files: These serve a variety of purposes, and their usage will be called out in the following sections.
- A `javadoc` directory: This contains the output of the javadoc tool run over all of the Java code in the Parliament code base. In some cases this documentation is nicely done, and in others it clearly needs work.
- A `License` directory: This contains the license for Parliament as well as the licenses for all of the components that are used by Parliament.

The sections below show how to use these artifacts to deploy Parliament in several different usage modes.

2.4 Alternate Server Configurations

The most common way to deploy Parliament is to combine it with Jena, Joseki, and Jetty to create a server application. This scenario is discussed above in Section 2.1. This section discusses other ways to deploy Parliament as a server.

2.4.1 Parliament with Jena, Joseki, & Tomcat

The Parliament distribution contains a war file that can be deployed in any servlet container, such as Apache Tomcat or Glassfish. This section contains pointers for deploying Parliament in a servlet container other than the provided Jetty.

1. First acquire and install Apache Tomcat[4] according to its own installation instructions. This will result in a directory containing your Tomcat installation, which, in keeping with the customary nomenclature of Tomcat, we will call `CATALINA_HOME` here.

2. Create a directory for the native Parliament binaries. This can reside anywhere on your machine, but just to be concrete we will call this directory `CATALINA_HOME/bin/parliament`.

3. From your binary distribution of Parliament, choose one of the directories of native build products that is appropriate for your operating system. (Remember, these are the directories that are named after the compiler that produced them — see Section 2.3.) Depending on the operating system, there may be some additional considerations:

    - On most platforms, you will need to further choose between 32-bit and 64-bit builds. On such platforms, these reside in sub-directories `32` and `64` within the compiler directory. You should use the 64-bit version for knowledge bases with more than 5 to 10 million statements. (Note that 64-bit Parliament cannot run within a 32-bit version of Java.)

    - The Macintosh build contains universal binaries for the x64 and x86 hardware architectures. The `-d32` and `-d64` command line switches to the `java` command select 32-bits and 64-bits, respectively.

    - If you have chosen to run binaries compiled with a version of Visual C++ on a non-development machine, then you may need to install the corresponding run-time libraries. You can find installers for these in the `RedistributablePackages` subdirectory of your general distribution of Parliament.

4. Once you have chosen your build directory, copy all of the files from it into the `CATALINA_HOME/bin/parliament` directory.

[4] `http://tomcat.apache.org/`

5. In your favorite text editor, open the file

   ```
   CATALINA_HOME/bin/parliament/ParliamentConfig.txt
   ```

   and change the following line:

   ```
   kbDirectoryPath        = .
   ```

   to point to a directory suitable for storing your knowledge base. In an operational deployment of Parliament, this is often set to a large drive dedicated to the storage of the knowledge base, such as a RAID array, NAS, or SAN. For testing purposes, set it to a relative path, such as:

   ```
   kbDirectoryPath        = data
   ```

   This stores the knowledge base in `CATALINA_HOME/bin/data`.

6. If you have not already done so while installing Tomcat, create a file called `setenv.sh` in `CATALINA_HOME/bin` and within it set the environment variable `CATALINA_OPTS` like so:

   ```
   MIN_MEM=128m
   MAX_MEM=512m
   PARLIAMENT_BIN="$CATALINA_HOME/bin/parliament"
   export CATALINA_OPTS="-Xms$MIN_MEM -Xmx$MAX_MEM"
   export PARLIAMENT_CONFIG_PATH=$PARLIAMENT_BIN/ParliamentConfig.txt
   export DYLD_LIBRARY_PATH=$PARLIAMENT_BIN
   export LD_LIBRARY_PATH=$PARLIAMENT_BIN
   ```

   On Windows, the file name should be `setenv.bat` and its content should look like this:

   ```
   set MIN_MEM=128m
   set MAX_MEM=512m
   set PARLIAMENT_BIN=%CATALINA_HOME%\bin\parliament
   set CATALINA_OPTS=-Xms%MIN_MEM% -Xmx%MAX_MEM%
   set PARLIAMENT_CONFIG_PATH=%PARLIAMENT_BIN%\ParliamentConfig.txt
   set PATH=%PARLIAMENT_BIN%;%PATH%
   ```

   Note that you can control the amount of memory set aside for the Java virtual machine by changing `MIN_MEM` and `MAX_MEM` in the scripts above. While it is important to allow the JVM sufficient memory, it is also important to realize that Parliament is native code, and therefore does not use the Java heap. Java programmers are often inclined to set `MAX_MEM` close to the total memory of the machine, but this will starve Parliament of the memory it needs to achieve good performance. The default values given above, 128 MB and 512 MB, are reasonable values for machines with 4 to 8 GB of total memory.

7. From the Parliament general distribution, copy `parliament.war` into Tomcat's appbase directory. This location defaults to the directory `CATALINA_HOME/webapps`.
8. Start Tomcat according to its documentation. For instructions on using your Parliament server, see Section 2.1.2.
9. If you redeploy Parliament into a running Tomcat instance, the server will need to be restarted.
10. *Please note:* When you want to shut down the KB, it is important to shut it down gracefully. Otherwise, the Parliament files will not be flushed to disk before they are closed, and they will almost certainly be badly corrupted. If you run Tomcat as a Windows service or UNIX Daemon, this is generally not a problem, because the operating system sends a shut-down message at the appropriate time. If, however, you run Tomcat explicitly via its startup script, you will need to shut it down yourself with Tomcat's stop script.

Glassfish Notes

When deploying Parliament to Glassfish, note the following:

- Glassfish sets it's own `java.library.path` property. The Parliament libraries must either be copied into `«Glassfish install dir»/lib` or you must alter the Glassfish configuration such that the libraries are available on `java.library.path`.
- By default, Parliament stores it's data files in the current working directory. In the case of Glassfish, this will be `«Glassfish install dir»/domains/domain1/config`. To choose a different location, change the setting `kbDirectoryPath` in the configuration file, `ParliamentConfig.txt`, to the location (absolute path) where you would like your files to reside.
- If Parliament is redeployed into a running instance of Glassfish, the server will need to be restarted.

2.4.2 Parliament with Sesame 1.2 and Tomcat

To create a knowledge base server based on Parliament, Sesame 1.2.x,[5] and Tomcat, do the following:

1. Install Apache Tomcat 5.5 or greater.[6]
2. Install Sesame 1.2.x.[7]
3. Deploy Sesame into Tomcat. Instructions on how to do this can be found in the file `doc/users/usersguide.html`, within the Sesame 1.2.x distribution, under section 2.2.
4. Copy the files `ParliamentSail.jar` and `Parliament.jar` into the directory webapps/sesame/WEB-INF/lib under the Tomcat installation.
5. From your binary distribution of Parliament, choose one of the directories of native build products that is appropriate for your operating system. (Remember, these are the directories that are named after the compiler that produced them — see Section 2.3.) Depending on the operating system, there may be some additional considerations:
    - On most platforms, you will need to further choose between 32-bit and 64-bit builds. On such platforms, these reside in sub-directories `32` and `64` within the compiler directory. You should use the 64-bit version for knowledge bases with more than 5 to 10 million statements. (Note that 64-bit Parliament cannot run within a 32-bit version of Java.)
    - The Macintosh build contains universal binaries for the x64 and x86 hardware architectures. The `-d32` and `-d64` command line switches to the `java` command select 32-bits and 64-bits, respectively.
    - If you have chosen to run binaries compiled with a version of Visual C++ on a non-development machine, then you may need to install the corresponding run-time libraries. You can find installers for these in the `RedistributablePackages` subdirectory of your general distribution of Parliament.

[5]Note that Sesame 2.0 and greater are not yet supported. Anyone willing to contribute work towards a Sesame 3.0 integration is most welcome to do so!

[6]`http://tomcat.apache.org/`

[7]`http://www.openrdf.org/`

6. Once you have chosen your build, copy the following files from it to the `bin` sub-directory of your Tomcat installation:
   - All of the DLL's or shared libraries — there should be two. One is Parliament itself, and the other is Berkeley DB. The exact filenames differ due to platform-specific naming conventions.
   - `ParliamentAdmin` (`ParliamentAdmin.exe` on Windows)
   - `ParliamentConfig.txt`
7. Add the XML fragment in Figure 2.3 to the `<repositorylist>` element found in Sesame's configuration file, `webapps/sesame/WEB-INF/system.conf`. You should replace `C:/MyKbDirectory/` with your own directory — this is the location where Parliament will create its data files, and the directory must exist before starting Sesame. (You should probably also change the `id` and `title`.)

```
<repository id="MyRepository">
  <title>My RDF Repository</title>
  <sailstack>
    <sail class=
      "com.bbn.parliament.sesame.sail.KbSyncRdfSchemaRepository"/>
    <sail class=
      "com.bbn.parliament.sesame.sail.KbRdfSchemaRepository">
      <param name="dir" value="C:/MyKbDirectory/"/>
    </sail>
  </sailstack>
  <acl worldReadable="true" worldWritable="true"/>
</repository>
```

Figure 2.3: Configuring Parliament in Sesame's `system.conf` File

8. Customize the Parliament configuration file `ParliamentConfig.txt` as necessary. Note that the `kbDirectoryPath` setting is ignored in favor of the `dir` parameter in the `system.conf` file (see Figure 2.3).
9. Start Tomcat. Tomcat uses port 8080 by default, so you may see an error message if this port is already in use.
10. Connect to `http://localhost:8080/sesame/` with your web browser, and select "My RDF Repository".

From this point on, simply use Sesame as you normally would, and all RDF storage and access will be redirected to the underlying Parliament instance.

One final cautionary note: When you want to shut down the KB, it is important to shut it down gracefully. Otherwise, the Parliament files will not be flushed to disk before they are closed, and they will almost certainly be badly corrupted. Consult the Tomcat documentation for shutdown instructions.

2.5 Using Parliament In-Process

2.5.1 Parliament In-Process with Jena

Jena is a popular toolkit for manipulating RDF data in Java programs. The central concept in the Jena class library is the Model interface. An object of type Model represents a graph of RDF data, and features many methods for manipulating that data. With just a few lines of code, you can create a Jena Model that is backed by an instance of Parliament, as demonstrated by Figure 2.4. The

```
import java.io.File;
import com.bbn.parliament.jena.graph.KbGraph;
import com.bbn.parliament.jena.graph.KbGraphFactory;
import com.bbn.parliament.jni.Config;
import com.hp.hpl.jena.rdf.model.Model;
import com.hp.hpl.jena.rdf.model.ModelFactory;

private Model createParliamentModel()
{
  KbGraph baseGraph = KbGraphFactory.createDefaultGraph();
  return ModelFactory.createModelForGraph(baseGraph);
}
```

Figure 2.4: Creating a Jena Model backed by Parliament

`createParliamentModel` method returns a Jena model backed by a Parliament instance. With this model, you can do any of the things you normally do with a Jena model, such as call `read` on it to load a file into it, or query it.

The `useParliamentModel` method shown in Figure 2.5 demonstrates how to call `createParliamentModel`. The important thing to note here is the try-finally construct. This guarantees that the model is closed properly, even if an exception is thrown. Closing the model is crucial, because if you don't, the Parliament files will not be flushed to disk before they are closed, and they will almost certainly be badly corrupted.

```
import com.hp.hpl.jena.rdf.model.Model;

void useParliamentModel()
{
  Model kbModel = createParliamentModel();
  try
  {
    // Use the model according to the Jena documentation.
    // For instance, to issue a SPARQL query:
    QueryExecution exec = QueryExecutionFactory.create(
      sparqlQueryString, kbModel);
    for (ResultSet rs = exec.execSelect(); rs.hasNext(); rs.next())
    {
      // Do something useful with the result set here
    }
  }
  finally
  {
    if (kbModel != null && !kbModel.isClosed())
    {
      kbModel.close();
      kbModel = null;
    }
  }
}
```

Figure 2.5: Using a Jena Model backed by Parliament

Naturally, there is some setup required to make this work:

1. You will need `JenaParliament.jar` and `Parliament.jar` from your binary distribution, along with the Jena jar files, which you can download from the Jena web site.[8] Place all of these jar files in a convenient location and ensure that they are added to your class path. (If you are using Eclipse, add them to the build path.)

2. From your binary distribution of Parliament, choose one of the directories of native build products that is appropriate for your operating system. (Remem-

[8] http://openjena.org/

ber, these are the directories that are named after the compiler that produced them — see Section 2.3.) Depending on the operating system, there may be some additional considerations:

- On most platforms, you will need to further choose between 32-bit and 64-bit builds. On such platforms, these reside in sub-directories `32` and `64` within the compiler directory. You should use the 64-bit version for knowledge bases with more than 5 to 10 million statements. (Note that 64-bit Parliament cannot run within a 32-bit version of Java.)
- The Macintosh build contains universal binaries for the x64 and x86 hardware architectures. The `-d32` and `-d64` command line switches to the `java` command select 32-bits and 64-bits, respectively.
- If you have chosen to run binaries compiled with a version of Visual C++ on a non-development machine, then you may need to install the corresponding run-time libraries. You can find installers for these in the `RedistributablePackages` subdirectory of your general distribution of Parliament.

3. Once you have chosen your build, copy the following files from it to a convenient location (e.g., the same directory used for the jar files above):
   - All of the DLL's or shared libraries — there should be two. One is Parliament itself, and the other is Berkeley DB. The exact filenames differ due to platform-specific naming conventions.
   - `ParliamentAdmin` (`ParliamentAdmin.exe` on Windows)
   - `ParliamentConfig.txt`
4. Ensure that the binaries are on your operating system's library path. On Windows this is the `Path` environment variable. Mac OS uses the environment variable `DYLD_LIBRARY_PATH`, and Linux uses `LD_LIBRARY_PATH`. Many flavors of UNIX also use `LD_LIBRARY_PATH`, though some differ. Consult your operating system documentation to be sure you use the correct variable.
5. Customize the Parliament configuration file `ParliamentConfig.txt` as necessary. In particular, the kbDirectoryPath setting defaults to the current working directory ".". To place your Parliament KB files in a different location, simply change this setting to the directory of your choice. This is especially useful for placing the KB files on a drive array, or for maintaining

several KB's and switching between them.

6. Set the `PARLIAMENT_CONFIG_PATH` environment variable to the configuration file's path. On Windows, if this is not set the file will be loaded from the same directory as the Parliament DLL, which is often a useful default. On other platforms there is no default, and so this environment variable must be set.

If `createParliamentModel` throws an `UnsatisfiedLinkError` exception, see Section 2.8 for possible resolutions.

2.5.2 Parliament In-Process with Sesame 1.2

[Yet to be written]

2.5.3 Parliament Direct, via Java

[Yet to be written]

2.5.4 Parliament Direct, via C++

[Yet to be written]

2.6 Configuring Indexes

[Yet to be written]

2.7 The ParliamentAdmin Utility

[Yet to be written]

2.8 Troubleshooting

Because Parliament is most often used within a Java environment, the most common problem is the dreaded "Unsatisfied Link Error". This error is generated by the Java Virtual Machine (JVM) when the Java code requests the loading of a DLL,[9] but the JVM is unable to load that DLL. There are quite a few reasons why you may encounter an unsatisfied link error, and unfortunately, the JVM is not very good about generating an illuminating error message. So, should you encounter this problem, here are some things to keep in mind:

- Double-check that you have followed all the deployment instructions correctly for your chosen mode of deployment.
- Make sure that your system is up-to-date with respect to patches. For Windows, this means a visit to the Windows Update web site.[10] Most other operating systems have a similar facility.
- Be sure that a 64-bit build of Parliament is loaded by a 64-bit version of Java, and that a 32-bit build of Parliament is loaded by a 32-bit version of Java. On Macintosh, this is not a problem, because Parliament is built as a universal binary.
- When the JVM loads the Parliament DLL, Parliament itself loads several other DLL's. Even if the JVM is able to load Parliament, an unsatisfied link error will result if Parliament cannot load one of its subsidiary DLL's. Furthermore, the error message accompanying the unsatisfied link error usually does not indicate which DLL caused the load failure. The subsidiary DLL's fall into the following three categories:
  - The Berkeley DB and Boost DLLs: In the deployment procedures detailed in this document, these DLLs should reside in the same location as Parliament itself.
  - The C and C++ run-time libraries: On Unix-like systems these should be available automatically. On Windows you may need to install the Visual Studio run-time libraries. You can find installers for these in the

[9] On UNIX-like systems, these are called "shared libraries", but for the sake of brevity, we will use DLL (Dynamically Linked Library) as a generic term for all operating systems.

[10] `http://windowsupdate.microsoft.com/`

`RedistributablePackages` subdirectory of your binary distribution of Parliament.

  - Various operating system DLL's: These are rarely a problem, because the OS needs to be able to find these to function.

- The Java system property `java.library.path` is a list of paths that Java searches when loading native libraries. This suggests that all of the operating system-specific rules for locating DLLs can be circumvented by providing a suitable definition for this system property. Unfortunately, when Java loads a library that itself loads other libraries (which Parliament does), Java can consult `java.library.path` only when loading the top-level library, because the loading process for its dependents is managed entirely within the operating system. Thus, this property does not solve the problem for Parliament.

- Running `ParliamentAdmin` helps to diagnose unsatisfied link errors, because this tool also loads the Parliament DLL, but it does so without involving Java (because `ParliamentAdmin` is written in C++). This technique can be used to isolate the error to either the operating system level (when `ParliamentAdmin` fails to load the DLL) or to the Java level (when `ParliamentAdmin` succeeds). Even running "`ParliamentAdmin -v`" to get the version requires loading the Parliament DLL, so this is an easy test.

- Multiple Java installations can confuse the loading process. The scripts `StartParliament.sh` and `StartParliament.bat` find Java by simply invoking the command `java` and relying on the operating system to locate the installation. In these cases, the `JAVA_HOME` environment variable is not used, unless the operating system uses it. On Windows, the process by which the `java` command finds and loads the JVM is quite complex. The following web page sheds some light on how you can manage this process:

  http://mindprod.com/jgloss/javaexe.html#MULTIPLES

  In the case of the `InstallParliamentService` script, the situation is quite different. The `java` command is not used at all, and instead the service executable directly loads the `jvm.dll` library. This is located by using the `JAVA_HOME` environment variable at the time the service is installed. (This means that if you upgrade your Java installation after installing the service, you will have to uninstall and reinstall the service in order for it to find the `jvm.dll` library.)

- The process by which Windows searches for DLL's is somewhat complex, so understanding it can often help pinpoint the problem. The search process is as follows:
  - The Windows system directory
  - The Windows directory
  - The directory where the executable module for the current process is located
  - The current directory
  - The directories listed in the `PATH` environment variable

  Note that placing DLL's in the Windows or Windows system directories is strongly discouraged.

  Both the graphical tool Dependency Walker[11] and the command-line tool `dumpbin` (part of the Visual Studio installation) show from where Windows is trying to load subsidiary DLL's. (Note that the operating system DLL `advapi.dll` itself loads many other libraries, some by magic, and that it therefore confuses Dependency Walker, which reports that `X` and `Y` are missing. These error messages are, however, red herrings and should be ignored.)

- UNIX-like systems find shared libraries by searching the directories in the library search path environment variable. The name of this variable differs by system: It is `DYLD_LIBRARY_PATH` on Macintosh, whereas on Linux it is `LD_LIBRARY_PATH`. Other UNIX-like systems may have still other names for this variable. On most UNIX-like systems the command `ldd` will show from where the operating system is trying to load subsidiary DLL's. On Macintosh, the command "`otool -L`" performs a similar function.

- A debug build of Parliament will not work on Windows unless the corresponding version of Visual Studio is installed. This is because the debug versions of the C and C++ run-time libraries are included only with Visual Studio.

- More rarely, unsatisfied link errors can result from a bad build of Parliament that causes the symbols exported from the DLL to be named incorrectly. If this happen, the JVM will succeed in loading the DLL, but fail to find

[11] `http://dependencywalker.com/`

the entry points it needs. This results in an unsatisfied link error that is indistinguishable from the cases where the JVM cannot load the DLL at all. This condition is more likely to occur on Windows, and usually has something to do with the symbol `BUILDING_KBCORE` not being defined on the compiler command line.

Chapter 3

Building Parliament

Parliament is a cross-platform, mixed-language library. It's core is written in portable C++, but it also has a Java interface. As a result of both the cross-platform and multi-language requirements, the build infrastructure for Parliament requires a little bit of work to configure. This chapter is your guide through that process.

Parliament's build infrastructure has two main parts. The top-level portion is based on ant, an XML-based build tool widely used in the Java development community. This portion of the infrastructure builds the Java half of the Parliament code base, and it also invokes the second portion, which is based on Boost.Build. Boost.Build is a system that is well-adapted to building C++ code. It has the advantages of being portable and much simpler to use than make files. It is also the standard build system of the Boost project, whose libraries are used by the C++ portion of Parliament.

This chapter will step through the libraries and tools that Parliament depends upon and show you how to configure them on your system. At the end of this chapter, you should have a working copy of the Parliament source code from which you can build Parliament binaries.

3.1 Platforms and Prerequisites

You will need to acquire and install one or more appropriate compilers for each operating system on which you wish to build. Parliament has been tested on the platform and compiler combinations shown in Table 3.1. Note that the last column shows the corresponding Boost.Build toolset name, which will be used extensively in the sections below as we configure the Parliament build infrastructure.

Operating System	**Compiler**	**Toolset**
Windows (32- and 64-bit)	Visual Studio 2013	msvc-12.0
MacOS 10.10.1	Xcode 6.1.1	clang
Ubuntu 14.10 (32- and 64-bit)	GCC 4.9.1	gcc
Centos 6.5 (64-bit)	GCC 4.4.7	gcc

Table 3.1: Supported Platforms and Compilers

Windows versions 7 and above are supported. The 64-bit build has been tested only on AMD-64 or EM64T hardware (often collectively known as x64), but not Itanium (IA64). Parliament's capacity is much higher when running as a 64-bit process,[1] so one of those compilers is highly recommended. Note that Visual Studio's 64-bit compilers are *not* installed by default, so you will have to choose the "Custom" option during installation and select them explicitly. You can also re-run the Visual Studio installer to add the 64-bit tools to an existing installation.

On Macintosh, Parliament builds as a universal binary (for the x64 and x86 architectures) using Apple's Xcode development tools.

On Linux, Parliament has been explicitly tested only on Ubuntu 13.10. On that distribution, Parliament supports both 32- and 64-bit builds (x86 and x64). Note that you may need to install extra packages to enable cross-compilation. For instance, the author installed the 64-bit edition of Ubuntu, and then added the "g++-multilib" package.

Parliament assumes the presence of the Java Developer Kit (JDK), version 7 or above. Furthermore, you will need a 64-bit JVM in order to run a 64-bit build of Parliament.

- On Windows, you must download and install the JDK from Oracle. The 32-bit and 64-bit versions are separate downloads and installations.

[1]On 32-bit Windows Parliament runs out of virtual address space after storing 5 to 10 million statements.

- On Macintosh, you must download and install the JDK from Oracle.
- On Linux, you may need to install one or more packages, depending on your particular distribution.

On Windows and Linux, choosing whether to invoke 32- or 64-bit Java is straightforward — just invoke the appropriate version of the `java` executable. On Macintosh, there are several ways to switch between 32- and 64-bit Java because MacOS uses universal binaries. One is to use the `-d32` and `-d64` command line switches to the `java` command. See the Java documentation for details.

You will need Apache Ant version 1.7.0 or later. On Macintosh this is built-in. Windows users can acquire Ant from the Apache web site,[2] and Linux users can typically get ant through their package manager.

Finally, you need to install a client for the Subversion version control system.[3] There are several different clients, depending on your operating system and your preferred mode of usage, but any of them are fine for our purposes here. On Macintosh the command line client is built-in.

Once you have a working Subversion client, you can check out a working copy of the Parliament code base like so:

```
svn --username «name» co
    https://projects.semwebcentral.org/svn/parliament/trunk «dir»
```

Change `«dir»` to a convenient location on your local machine. For anonymous access, change `«name»` to "anonsvn" and use the password "anonsvn". For project developer access via DAV, substitute your user ID for `«name»` and enter your site password when prompted.

3.2 Configuring Eclipse

The Parliament code base includes Eclipse projects for all of the Java and C++ code. These are useful for inspecting and editing code, but it is important to note that they are not the official build mechanism. In fact, as of this writing, the C++ Eclipse projects will not build at all. This may be corrected in the future, but the

[2] http://ant.apache.org/
[3] http://subversion.apache.org

JNI interface between Java and native code makes this complex. Therefore the C++ projects are merely an editing convenience.

To setup Eclipse, you need the Kepler version or later of the Eclipse IDE for Java Developers, plus the Eclipse C/C++ Developers Toolkit (CDT) plug-ins. One way to acquire this set of components is to download the Eclipse IDE for Java Developers,[4] install it, run it, and then use the "Install New Software" menu item to download and install the CDT. The procedure for this changes between Eclipse releases, but you can find instructions on the CDT web site.[5]

To use Eclipse with your Parliament working copy, first choose (or create) an Eclipse workspace. Then import all existing projects from within your Parliament working copy. To do so, choose Import from the File menu and select "Existing Projects into Workspace" under the General category. Press the Next button, and enter the root directory of your Parliament working copy in the "Select root directory" box. Press the Select All button, make sure that "Copy projects into workspace" is unchecked, and press the Finish button. At this point all of the Parliament projects will be displayed in the Package Explorer view.

3.3 Building Berkeley DB

Parliament uses Berkeley DB (often abbreviated BDB), an embedded database manager from Oracle.[6] Because Parliament is open source, this use of Berkeley DB also falls under an open source license. The following procedures are based on BDB version 6.1.19.

3.3.1 Building BDB for Windows

The build infrastructure for Berkeley DB is not particularly friendly for Windows. Therefore, the Parliament source code repository includes pre-built Berkeley DB libraries (both 32- and 64-bit) for all supported versions of Visual Studio.[7] If you

[4] `http://www.eclipse.org/`

[5] `http://www.eclipse.org/cdt/`

[6] `http://www.oracle.com/us/products/database/berkeley-db/`

[7] Note that Oracle does provide an already-compiled distribution, but it includes only a 32-bit build.

need to update the pre-built libraries, e.g., for a new version of Berkeley DB or to build with a different compiler, see Appendix A.

Define the following environment variables so that the Parliament build infrastructure can find the libraries:

```
BDB_VERSION=6.1
BDB_HOME=«dir»\lib\bdb
```

where «`dir`» is the absolute path of your Parliament working copy.

3.3.2 Building BDB for Macintosh

On Macintosh, Berkeley DB follows the usual pattern of software based on the autoconf/automake/libtool suite. Specifically, expand the BDB distribution archive file, and change to the `build_unix` subdirectory. Then issue the following commands:

```
env CFLAGS="-arch x86_64 -arch i386" ../dist/configure
make
sudo make install
```

The `CFLAGS` setting above causes the build to produce universal binaries. You can tidy up after the build with the command `make realclean`. Once you have built and installed Berkeley DB, define the following environment variables so that the Parliament build infrastructure can find the libraries:

```
BDB_VERSION=6.1
BDB_HOME=/usr/local/BerkeleyDB.6.1
```

3.3.3 Building BDB for Linux

On Linux, Berkeley DB follows the usual pattern of software based on the autoconf/automake/libtool suite. Here we modify that procedure slightly to create both 32- and 64-bit builds. First, decompress and un-archive the BDB distribution, and change to the `build_unix` subdirectory. Next decide where you want to install Berkeley DB. The default location is `/usr/local/BerkeleyDB.6.1`, but whatever location you choose will be called «`dir`» in the instructions below. Once you have chosen this location, issue the following commands:

```
env CFLAGS="-m32" ../dist/configure --prefix=«dir»/32
make
sudo make install
make realclean
env CFLAGS="-m64" ../dist/configure --prefix=«dir»/64
make
sudo make install
```

You can tidy up after the build with the command `make realclean`. Once you have built and installed Berkeley DB, define the following environment variables so that the Parliament build infrastructure can find the libraries:

```
BDB_VERSION=6.1
BDB_HOME=«dir»
```

3.4 Building the Boost Libraries

Since the Boost project is unfamiliar to many, here is an introduction taken from the Boost web site:[8]

> Boost provides free peer-reviewed portable C++ source libraries.
>
> We emphasize libraries that work well with the C++ Standard Library. Boost libraries are intended to be widely useful, and usable across a broad spectrum of applications. The Boost license encourages both non-commercial and commercial use.
>
> We aim to establish "existing practice" and provide reference implementations so that Boost libraries are suitable for eventual standardization. Ten Boost libraries are already included in the C++ Standards Committee's Library Technical Report (TR1) and in the new C++11 Standard. C++11 also includes several more Boost libraries in addition to those from TR1. More Boost libraries are proposed for TR2.

To get started, download the Boost source distribution (version 1.57.0 or later) from the Boost web site. Unpack this to a handy location on your disk. This location may be anywhere you like, but note that it is not temporary. We will call this location `BOOST_ROOT`, and you need to define an environment variable pointing there, such as

```
export BOOST_ROOT=~/boost_1_57_0
```

[8] http://boost.org/

or

```
set BOOST_ROOT=C:\boost_1_57_0
```

From Parliament's point of view, there are two primary components contained within `BOOST_ROOT`. The first (and most obvious) is the boost libraries themselves. Most of these are so-called "header-only" libraries, meaning that there is no precompiled library code or shared or dynamic library. All of the code of such libraries is referenced via `#include` directives and compiled along with the calling code. Such libraries are extremely convenient, because they require virtually zero setup. Parliament uses several Boost libraries that are not header-only. We will discuss how to build these libraries below.

The second major Boost component is Boost.Build. This is a cross-platform build system (located in the sub-directory `tools/build`) that is implemented in an interpreted language, and the language interpreter is a command line program called `b2`. The Boost community does not provide `b2` binaries. Rather, the Boost distribution contains a bootstrapping script that builds `b2` from source on your platform. At a command line, change to the `BOOST_ROOT` directory and issue the appropriate one of these commands:

`bootstrap.bat`	(on Windows)
`./bootstrap.sh --with-toolset=clang`	(on Macintosh)
`./bootstrap.sh`	(on Linux)

Once the script finishes, you will find the executable `b2` (or `b2.exe` on Windows) in `BOOST_ROOT`. Move this binary to any location on your path.

Building the Boost libraries using Boost.Build is relatively simple. To build the minimal set of libraries required for Parliament, follow the directions below for your platform.

3.4.1 Building Boost on Windows

Open a Command Prompt and change to the `BOOST_ROOT` directory. To build 32-bit libraries, issue the following command:

```
b2 -q --build-dir=msvc-32/build --stagedir=msvc-32/stage
--with-date_time --with-filesystem --with-log --with-regex
--with-system --with-test --with-thread
toolset=msvc-12.0 address-model=32
```

```
variant=debug,release threading=multi link=shared
runtime-link=shared stage
```

For 64-bit libraries, issue the following command:

```
b2 -q --build-dir=msvc-64/build --stagedir=msvc-64/stage
--with-date_time --with-filesystem --with-log --with-regex
--with-system --with-test --with-thread
toolset=msvc-12.0 address-model=64
variant=debug,release threading=multi link=shared
runtime-link=shared stage
```

Remember that you need to explicitly choose to install Microsoft's 64-bit compilers when you install Visual Studio. Failure to do so will result in mysterious error messages whose cause is not clear.

The build process may issue several warnings about missing components, such as Python and ICU. These warnings are innocuous.

When the build finishes, the libraries will be located in the directories `msvc-32/stage/lib` and `msvc-64/stage/lib`. The directories `msvc-32/build` and `msvc-64/build` contain the intermediate build products and can be deleted to save disk space.

3.4.2 Building Boost on Macintosh

Open a Terminal window, change to the `BOOST_ROOT` directory, and issue the following command:

```
b2 -q --build-dir=clang/build --stagedir=clang/stage
--layout=versioned --with-date_time --with-filesystem
--with-log --with-regex --with-system --with-test --with-thread
address-model=32_64 variant=debug,release threading=multi
link=shared runtime-link=shared stage
```

The build process may issue several warnings about missing components, such as Python and ICU. These warnings are innocuous.

When the build finishes, the libraries will be located under `clang/stage/lib`. The directory `clang/build` contains the intermediate build products and can be deleted to save disk space.

3.4.3 Building Boost on Linux

At a shell, change to the `BOOST_ROOT` directory. To build 32-bit libraries, issue the following command:

```
b2 -q --build-dir=gcc-32/build --stagedir=gcc-32/stage
--layout=versioned --with-date_time --with-filesystem
--with-log --with-regex --with-system --with-test --with-thread
address-model=32 variant=debug,release threading=multi
link=shared runtime-link=shared stage
```

For 64-bit libraries, issue the following command:

```
b2 -q --build-dir=gcc-64/build --stagedir=gcc-64/stage
--layout=versioned --with-date_time --with-filesystem
--with-log --with-regex --with-system --with-test --with-thread
address-model=64 variant=debug,release threading=multi
link=shared runtime-link=shared stage
```

The build process may issue several warnings about missing components, such as Python and ICU. These warnings are innocuous.

When the build finishes, the libraries you need will be located under `gcc-32/stage/lib` and `gcc-64/stage/lib`. The directories `gcc-32/build` and `gcc-64/build` contain the intermediate build products and can be deleted to save disk space.

3.5 Configuring Boost.Build

Next we need to configure Boost.Build so that it can build Parliament. Start by defining the following environment variables:

- `BDB_VERSION`: As described above in Section 3.3.
- `BDB_HOME`: As described above in Section 3.3.
- `BOOST_VERSION`: The version of Boost (currently 1_57_0).
- `BOOST_ROOT`: As described above in Section 3.4.
- `BOOST_BUILD_PATH`: The sub-directory `tools/build` of `BOOST_ROOT`.

- `BOOST_TEST_LOG_LEVEL`: Controls the volume of output from the unit tests for Parliament's C++ code. Possible values and their meanings are listed in Table 3.2.

Value	**Meaning**
`all`	report all log messages
`success`	the same as all
`test_suite`	show test suite messages
`message`	show user messages *(useful default)*
`warning`	report warnings issued by user
`error`	report all error conditions
`cpp_exception`	report uncaught C++ exceptions
`system_error`	report system-originated non-fatal errors
`fatal_error`	report only fatal errors
`nothing`	do not report any information

Table 3.2: Possible Values of `BOOST_TEST_LOG_LEVEL`

- `JAVA_HOME`: The location of your JDK installation, which must be version 7 or higher.

- `JNI_HEADERS`: On MacOS only, set this to the location of your JDK include directory, where the JNI header files reside. If you are using Apple's Java 6, set this to `/System/Library/Frameworks/JavaVM.framework/Headers`. For Oracle's Java 7, use `$JAVA_HOME/include`.

Next we create two Boost.Build configuration files, `site-config.jam` and `user-config.jam`. Boost.Build reads these files on startup. The two are separate so that the first one can be installed and maintained by a system administrator, and the second by the individual user. These files can be placed in a number of locations. Table 3.3 explains where Boost.Build searches to find these files.

Some people prefer to keep these files together with their Boost.Build installation, placing them in `BOOST_BUILD_PATH`. There are example `site-config.jam` and `user-config.jam` files in that directory, and so you will have to replace (or rename) them if you choose this option. Others prefer to separate the configuration files from the Boost.Build installation so that they can update to a new version of Boost.Build without having to first save `site-config.jam` and `user-config.jam` and then restore them after the update is complete. On Windows, using the `HOME`

OS	`site-config.jam`	`user-config.jam`

Unix-like:	`/etc`
	`$HOME`
	`$BOOST_BUILD_PATH`
		`$HOME`
		`$BOOST_BUILD_PATH`

Windows:	`%SystemRoot%`
	`%HOMEDRIVE%%HOMEPATH%`
	`%HOME%`
	`%BOOST_BUILD_PATH%`
		`%HOMEDRIVE%%HOMEPATH%`
		`%HOME%`
		`%BOOST_BUILD_PATH%`

Table 3.3: Boost.Build Search Paths for Configuration Files

location requires defining the environment variable `HOME`, because Windows does not define it by default.

Within your working copy of the Parliament repository, in the directories `doc/MacOS`, `doc/Windows`, and `doc/Linux`, you will find example configuration files for Macintosh, Windows, and Linux respectively that you can copy and customize. The most important customization that you need to make is to remove (or comment out) any lines in `user-config.jam` for compiler versions that you have not installed.

3.6 Building Parliament Itself

You are now ready to build Parliament. To do so, issue the command `ant` from the root directory of your Parliament working copy — *but don't try this until you have read the next couple of paragraphs.* This command will build the entire repository, including both native and Java code, and create a distribution-ready package in the `targets/artifacts` directory. (See Section 2.3 for a discussion of this directory.) The `ant clean` command will delete all build products.

The file `build.properties`, located in the root of your working copy, controls the Parliament build. This file does not exist by default. If the build does not find it, it uses `build.properties.default` instead. The latter contains the build options used to create an official release of Parliament, which typically includes several different release builds. To build a single debug variant, copy `build.properties.default` to `build.properties` and then customize the latter. The file itself contains instructions. Please customize `build.properties.default` directly only if you intend to change the official release build options.

The `build.properties` file also contains an option that disables the native code unit tests, `skipNativeUnitTest`. This is important when cross-compiling, e.g., building 64-bit binaries on 32-bit Windows. If you use this option, be sure to re-enable the unit tests before you commit any changes to the Subversion repository.

For more targeted builds, ant can be run from many sub-directories in your working copy. Here is a road map to the various sub-projects:

- `jena/JenaGraph`: Enables Jena to use Parliament for storing models
- `jena/JosekiExtensions`: Extensions to Joseki that, together with JenaGraph, create a SPARQL endpoint on top of Parliament
- `jena/JosekiParliamentClient`: A client-side Java library for communicating with a Joseki-Parliament SPARQL endpoint
- `jena/SpatialIndexProcessor`: A JenaGraph add-on for processing spatial queries efficiently
- `jena/TemporalIndexProcessor`: A similar add-on to speed up temporal queries
- `LUBM`: A customized version of the Lehigh University Benchmark[9] for easy performance testing of Parliament
- `Parliament`: The native code at the heart of Parliament and its JNI interface
- `Sesame/CSameAsSail`: A Sesame[10] plug-in that enables Sesame 1.2 to perform same-as inferencing
- `Sesame/LuceneSail`: A Sesame 1.2 plug-in to integrate full-text search with SeRQL queries
- `Sesame/ParliamentSail`: An integration that enables Sesame 1.2 to use Parliament as a storage layer[11]
- `Sesame/SameAsSail`: Another plug-in that enables same-as inferencing in Sesame 1.2

[9] `http://swat.cse.lehigh.edu/projects/lubm/`

[10] `http://www.openrdf.org/`

[11] Anyone willing to contribute work towards a Sesame 3.0 integration is most welcome!

- `Sesame/Swrl2Sesame`: SWRL-to-Sesame conversion

When working on the native code portions of Parliament, it can be useful to run the `b2` portion of the build directly. To do so, change directory to `KbCore` (for the Parliament DLL itself), `AdminClient` (for the command line interface to Parliament), or `Test` (for the unit tests). These directories are located within the `Parliament` sub-directory of your working copy. Then issue the command

```
b2 -q «build-options»
```

Here «`build-options`» is a placeholder for one set of build options from `build.properties`, described above. The `-q` option causes `b2` to quit immediately whenever an error occurs, so that you do not have to scroll up through the build output to verify that the build was successful.

Appendix A

Building Berkeley DB for Windows

The build infrastructure for Berkeley DB is not particularly friendly for Windows. Therefore, the Parliament source code repository includes pre-built versions of Berkeley DB for Visual Studio, in both 32- and 64-bit flavors.[1]

This appendix is provided primarily to guide the developer who needs to update the pre-built libraries, e.g., for a new version of Berkeley DB or to build with a different compiler. If this does not apply to you, then you may skip this appendix.

1. Download the source code distribution for Berkeley DB and unzip it to a directory of your choice on your local disk. Unless otherwise specified, all paths mentioned below are relative to this location.
2. In Visual Studio 2013, open this solution file:

   ```
   build_windows\Berkeley_DB_vs2010.sln
   ```

   Allow the Visual Studio Conversion Wizard to convert the projects to its file format.
3. For each configuration-platform pair listed in Table A.1, choose "Build / Configuration Manager" from the menu and select that configuration and platform. Then, in the Solution Explorer, right-click the "db" project and choose "Project Only / Build Only db" from the menu.
4. Close Visual Studio.

[1]Oracle does provide an already-compiled binary distribution, but it includes only a 32-bit build.

Configuration	Platform
Debug	Win32
Release	Win32
Debug	x64
Release	x64

Table A.1: Visual Studio Configuration-Platform Pairs

5. Create the directory hierarchy shown in Figure A.1 at the root level of your Subversion working copy.

```
lib
└── bdb
    ├── include
    └── msvc-12.0
        ├── 32
        └── 64
```

Figure A.1: BDB Directory Hierarchy

6. Copy these files into the `msvc-12.0\32` directory created in Step 5 above:

```
build_windows\Win32\Debug\db\libdb61d.dll
build_windows\Win32\Debug\db\libdb61d.lib
build_windows\Win32\Debug\db\libdb61d.pdb
build_windows\Win32\Release\db\libdb61.dll
build_windows\Win32\Release\db\libdb61.lib
build_windows\Win32\Release\db\libdb61.pdb
```

7. Copy these files into the `msvc-12.0\64` directory created in Step 5 above:

```
build_windows\x64\Debug\db\libdb61d.dll
build_windows\x64\Debug\db\libdb61d.lib
build_windows\x64\Debug\db\libdb61d.pdb
build_windows\x64\Release\db\libdb61.dll
build_windows\x64\Release\db\libdb61.lib
build_windows\x64\Release\db\libdb61.pdb
```

8. Copy these files into the `include` directory created in Step 5 above:

```
build_windows\*.h
```

9. Commit the above changes to Parliament's Subversion repository.

10. Finally, delete the build directory that you created in Step 1 by unzipping the source code distribution.