Enhancing Client Honeypots with Grid Services and Workflows

by

David Stirling

A thesis
submitted to the Victoria University of Wellington
in fulfilment of the
requirements for the degree of
Master of Science
in Computer Science.

Victoria University of Wellington
2010

Abstract

Client honeypots are devices for detecting malicious servers on a network. They interact with potentially malicious servers and analyse the Web pages returned to assess whether these pages contain an attack. This type of attack is termed a 'drive-by-download'. Low-interaction client honeypots operate a signature-based approach to detecting known malicious code. High-interaction client honeypots run client applications in full operating systems that are usually hosted by a virtual machine. The operating systems are either internally or externally monitored for anomalous behaviour.

In recent years there have been a growing number of client honeypot systems being developed, but there is little interoperability between systems because each has its own custom operational scripts and data formats. By creating interoperability through standard interfaces we could more easily share usage of client honeypots and the data collected. Another problem is providing a simple means of managing an installation of client honeypots. Workflows are a popular technology for allowing end-users to co-ordinate e-science experiments, so these workflow systems can potentially be utilised for client honeypot management.

To formulate requirements for management we ran moderate-scale scans of the `.nz` domain over several months using a manual script-based approach. The main requirements were a system that is user-oriented, loosely-coupled, and integrated with Grid computing—allowing for resource sharing across organisations.

Our system design uses Grid services (extensions to Web services) to wrap client honeypots, a manager component acts as a broker for user access, and workflows orchestrate the Grid services. Our prototype wraps our case study—Capture-HPC—with these services, using the Taverna workflow system, and a Web portal for user access.

When evaluating our experiences we found that while our system design met our requirements, currently a Java-based application operating on our Web services provides some advantages over our Taverna approach—particularly for modifying workflows, maintainability, and dealing with failure. The Taverna workflows, however, are better suited for the data analysis phase and have some usability advantages. Workflow languages such as Taverna are still relatively immature, so improvements are likely to be made. Both of these approaches are significantly easier to manage and deploy than the previous manual script-based method.

Acknowledgments

I would like to express my gratitude to my supervisors—Dr. Peter Komisarczuk and Dr. Ian Welch—whose knowledge, support, and enthusiasm made this thesis possible. I must also thank Christian Seifert for initially getting me interested in client honeypots and providing a lot of assistance in this area. A special thanks goes to Kyle Chard for his help with everything Grid-related, reviewing this thesis, and generally making our shared office a more enjoyable place. Thanks goes to the other DSRG (Distributed Systems Research Group) members for their interest in my research and the feedback they provided. I am grateful to InternetNZ for funding the `.nz` domain client honypot study I have been involved in. I would finally like to acknowledge my family and friends for their encouragement and support during my MSc.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Client honeypots are measurement devices for detecting malicious servers on a network. They interact with potentially malicious servers and analyse the Web pages returned to assess whether it contains an attack. This type of attack is termed a ‘drive-by-download’. This has advantages over traditional defences, such as firewalls and antivirus software, are ineffective against these new threats [37] because the attack vectors delivered by the server maybe unknown to the firewalls or the antivirus software. Locating malicious servers and malware is a priority for helping protect consumers using the Web and for collecting malicious code for analysis by security researchers.

There are two types of client honeypot systems: low-interaction and high-interaction. Low-interaction client honeypots [37] operate a signature-based approach to detecting known malicious code. High-interaction client honeypots run client applications (such as Web browsers) in full operating systems that are usually hosted by virtual machines (VMs). These are either internally or externally monitored for suspicious state changes. Should malicious content be found, the state of the host operating system is now suspect so it must be reset to a clean state before it can interact with another server [38].

The client honeypot is an emerging research area. In recent years there have been a growing number of client honeypot systems being developed [44, 30, 35, 37, 45]. Between systems there is little interoperability, with each system having its own custom operational scripts and data formats. This means that there is a lack of code reuse for common tasks and users require detailed knowledge of these systems before they can use them.

Through the use of standardised Web service interfaces interoperability can be increased and implementation details abstracted, meaning that the usage of client honeypots and the data collected can be shared more easily. This can be expanded further with the use of Grid computing [15, 16], allowing secure co-ordination of heterogeneous resources located across multiple enterprises. This type of collaboration could potentially allow the collection and analysis of data on a scale that is many times what is currently possible.

Workflow systems can be used to orchestrate these Web services, creating processes out of a number of individual tasks. These can be graphically modelled allowing users with little programming knowledge to manipulate complex workflows. For example, a user could create a workflow that combined the speed of a low-interaction client honeypot with the accuracy of a high-interaction client honeypot; this could then be run in a workflow engine and shared with other researchers.

1.1 Research Questions

- How can we effectively create systems of different client honeypots to perform measurements and analysis?
- What is the best method to automate the various tasks that encompass client honeypot management?

1.2 Methodology

The main tasks that make up this research are:

- Gain experience by manually managing a client honeypot for moderate scale scans.

- Design a Grid-based architecture for managing client honeypots.
- Evaluate workflow systems for this type of activity.
- Create realistic client honeypot operational use cases and implement these as workflows.
- Implement a Web portal for browser based access to a client honeypot.
- Modify a client honeypot to use our architecture.
- Build a Java application for a comparative evaluation of our workflow system.

1.3 Research Contributions

The main contributions of this research are:

1. Creation of use cases to model system requirements of a client honeypot automation system, based on our experience of manually managing a system for moderate-scale scans over a period of several months.
2. Creation of Grid services for wrapping client honeypots. These can be used by workflow languages and standalone applications.
3. Implementation of workflows that use our services to model and control complex processes used in client honeypot operation.
4. Evaluation of Introduce and gRAVI—two recently developed Grid service authoring tools.
5. Evaluation of the Taverna workflow system for modelling and enacting client honeypot workflows, based on our experiences.

1.4 Overview of Chapters

Background and Related Work provides background information on relevant technologies: client honeypots, Grid computing and workflow systems; and presents related work in the area of instrument middleware.

Formulation of Requirements is focused on gathering requirements for an automated client honeypot system. From our experience of manually operating the Capture-HPC client honeypot for moderate-scale scans we created use cases and system requirements.

Design and Implementation describes how we designed a system to fulfil the main requirements, and implemented a prototype of this design using WSRF services, Taverna workflows and a Web portal. This chapter includes an evaluation of BPEL and Taverna, determining their suitability for our needs.

Evaluation covers the fulfilment of requirements, comparisons to other system designs (script-based and Java with Web services), general workflow usage in our domain, and an evaluation of service authoring tools. While our system design met the requirements, currently a Java-based application operating on our Web services provides some advantages over our Taverna approach. The Taverna workflows, however, are better suited for the data analysis phase.

Conclusion gives a summary of the thesis, answers to our research questions, a listing of our contributions, and directions for future work.

Appendices provide a sample Capture-HPC log file for a malicious site, a listing of shell scripts used during manual operation, the Web service interfaces, and source code for the Java & Web services alternative system design.

As part of this research we wrote two papers: [41] and [42]. The *Formulation of Requirements* and the *Design and Implementation* chapters expand upon the material covered in these papers.

Chapter 2

Background and Related Work

Section 2.1 gives background information on Client honeypots and describes the architecture of Capture-HPC, our case study client honeypot. Section 2.2 introduces Grid computing—specifically the service-oriented architecture, WSRF (Web Service Resource Framework), and other technologies and tools. Section 2.3 describes related work in the area of instrument middleware—CIMA and GRIDCC. Section 2.4 describes workflow systems, specifically BPEL (Business Process Execution Language) and Taverna. Section 2.5 introduces the Grid Enabled Internet Instruments (GEII) research project which we foresee our research being a part of. Finally, Section 2.6 provides a review of all the material.

2.1 Client Honeypots

Honeypots are dedicated networked security devices that are designed to lure malicious activity onto themselves, capturing all data associated with an attack. The device itself has no added value to a system, so any network traffic or new activity can be fully attributed to a security compromise—thus eliminating the issue of false positives found in other intrusion detection systems.

One of the major types of attack on the Internet is the client-side attack which targets a client application. Instead of a client attacking a server, the server delivers the attack to the client as part of the server's response to a client request. Common examples of these attacks are Web servers that attack Web browsers. When the Web browser requests content from a web server, the server returns a malicious page that attacks the browser. This could lead to arbitrary code being executed on the client machine—this type of attack is termed a 'drive-by-download'.

An example of such an attack would be a user with Internet Explorer 6 simply browsing to `http://wwww.blackmores.co.nz/`[1]. Unknown to the user, when the HTML content was downloaded and interpreted, a vulnerability in IE 6 was exploited (ie. a buffer overflow in some JavaScript code) and a piece of malicious code was executed outside the browser's security sandbox. This code downloads an executable to `C:\msntstza.exe` and launches this new process. This malicious process may be adware, spyware, or a botnet drone. The website owner may also be unaware that their site is serving drive-by-downloads if their site was compromised by a third party, eg. an attacker compromising the hosting company's Web servers.

Traditional defenses—such as firewalls and antivirus software—are ineffective against these new threats because the attack vectors delivered by the server maybe unknown to the firewalls or the antivirus software [38].

Client honeypots can be used to identify malicious servers on a network. They do so by generating a queue of server requests, issuing these requests to the servers one-by-one and consuming the response of the servers. After a response is consumed, the client honeypot can perform an analysis that determines whether the server is malicious or benign. Figure 2.1 shows the architecture of a generic client honeypot.

A low-interaction client honeypot will analyse the server's response and match the signature to a list of known attacks. If there is a match, the server will be classified as malicious. Honeyd Virtual Honeypot [30] is an example of a low-interaction client honeypot.

[1]This was an actual attack we captured, the log files are included in Appendix A

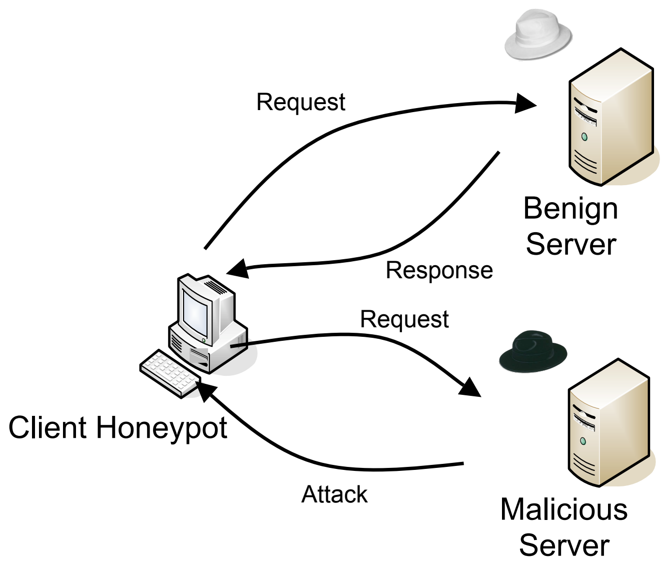


Figure 2.1: Client honeypot architecture [38].

A high-interaction client honeypot uses a more thorough approach. Full client operating systems and applications are used and the entire system is monitored for state changes. Classification is based on monitoring unauthorised state changes or actions occurring on the system after the client honeypot has interacted with a server. Client honeypots are dedicated machines and since no other activity is occurring on them, unauthorised state changes such as new processes, newly installed files, etc. can be detected by the client honeypot. Once state changes are detected and the classification has been made, the machine needs to be reset into a clean state before it can interact with another server [38]. The advantage of this approach is it can detect both known and unknown attacks, however the increased resources needed means this method is slower than a low-interaction client honeypot. Strider HoneyMonkey [45] is an example of a high-interaction client honeypot.

2.1.1 Capture Honeypot Client

Capture-HPC [35] is a high-interaction client honeypot being developed by Seifert et al. We have chosen Capture-HPC as a case study for this research. Figure 2.2 shows the architecture of Capture which runs on a group of networked computers. A set of URLs to be tested is passed to the Capture-server component, which (one-by-one) delegates these to Capture-client components to test. The CaptureClient instance is executed inside a Virtual Machine (VM) hosting Windows XP (a default installation and standard network security settings). Multiple VMs, each running a CaptureClient instance, can be concurrently run inside the same VM environment on one physical machine. When the CaptureClient receives the URL from the server it invokes Internet Explorer to visit the Web server and waits a set time (currently set at 10 seconds). The CaptureClient continuously records any state changes—by monitoring the registry, processes and file system—and sends this to the server for analysis (exclusion lists are used to identify benign system activity). CaptureServer then makes a decision on whether the Web server is malicious, in which case the server records the data and resets the VM to a clean snapshot ready for the next URL to test [36].

CaptureServer controls the CaptureClients by using the VMWare VIX API. The API interfaces with the VMWare Tools component which is installed in the Windows XP VM. This allows for control of the VM, both internally and externally, such as launching applications and powering on the VM respectively. During operation CaptureServer initially uses VIX to revert the VM to its saved snapshot and launches the `CaptureClient.exe` application (supplying its own IP address). The CaptureClient then creates a TCP connection to the server and waits for URLs to be passed to it. When it receives URLs it loads the browser to visit the page, reporting any state changes back over the TCP connection. If the state changes indicate the URL is malicious, CaptureServer resets the VM via VIX and the process begins again.

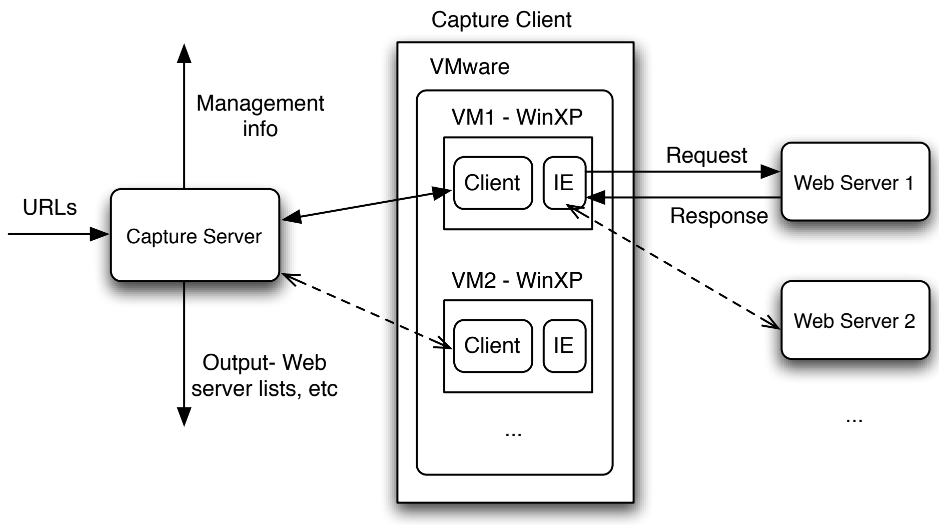


Figure 2.2: Capture-HPC architecture.

2.1.2 Data Capture and Analysis

To analyse a drive-by-download attack it is necessary to gather the data delivered from the Web server to the client. This cannot be done by simply saving the HTML of the Web page as this does not include any external JavaScript, iFrames, etc. This content will not necessarily be able to be retrieved at a later date because with the dynamic nature of the Web, content may change and sites can go down. A solution to this problem is to capture all of the network data using the pcap (packet capture) API. Capture-HPC uses the WinPcap utility to dump all network packets into a file for later analysis.

However, research has shown custom tools have to be developed to access and examine the dumped pcap data—decoding the network protocols—and once the information is extracted from the network data it does not allow behavioral analysis of the attack, limiting the ability to determine what happened on the attacked system [34]. The solution is to use HTTP and DNS caches to persistently store all traffic, creating a simple record/replay mechanism for data collection and behavioural analysis [34]. The Squid HTTP cache [47] and pdnsd DNS proxy [25] are suitable for this purpose—with only simple modifications to their configuration files required to prevent flushing and always force caching—thus allowing permanent data storage.

Using the data captured from tested URLs (log files and caches), data analysis techniques can be used to gather a better insight into malicious Web servers and the attacks they deliver. Analysis can be made of a specific URL to determine the geographic locations of the Web server and exploit server (using geo-IP services), characteristics of the HTML page, and the nature of the attack. By examining large sets of sample URLs the aforementioned URL-specific analysis can be used to draw inferences upon the experimental population. These samples, when taken over a period of time, allow trends to be analysed and possibly predictions to be made.

2.2 Grid Computing

Grid computing [16, 15] is an area of distributed computing, with the goal being "co-ordinated resource sharing and problem solving in dynamic, multi-institutional virtual organisations" [16]. The sharing that the Grid is concerned with is not the traditional file access and transfer, but rather access to computers, software, data, and other resources. This sharing is highly controlled with access determined by rules related to membership of virtual organisations (VOs). These VOs are formed from the heterogeneous resources within a single enterprise or multiple enterprises (eg. collaborating universities) and any external resource sharing or service provider relationships.

To help define what a Grid is, Foster provides a checklist [13] of requirements for a Grid system:

- *Coordinate resources that are not subject to centralised control*—a Grid integrates resources and users from different domains and addresses issues of security, policy, payments, etc. that cannot be centrally managed.

- *Using standard, open, general-purpose protocols and interfaces*—functionality such authentication, authorisation, resource discovery, and resource access all use multi-purpose protocols and interfaces.
- *Deliver nontrivial qualities of service*—a Grid allows its resources to be used in a coordinated fashion to deliver various qualities of service to meet complex user demands.

Client honeypots can currently be distributed but could potentially be enhanced further with Grid computing. Heterogeneous hardware located across multiple enterprises could be coordinated to provide client honeypots with data capture and analysis abilities. The Grid would also allow this data to be shared within a VO, which may include a number of universities from different countries. This type of collaboration could allow the collection and analysis of data on a scale that is many times what is currently possible. With this huge processing potential, almost real-time analysis may be feasible. In Section 2.2.3 we describe some of the specific Grid technologies that are applicable to client honeypots—including file transfer, storage and security.

2.2.1 Service Oriented Architecture

Service-oriented architecture (SOA) is widely used in the business sector for building large-scale information systems [8]. SOA defines standardised interfaces and protocols for exchanging data between services without requiring knowledge of the internal implementation. This level of abstraction and interoperability allows for complex applications to operate over multiple loosely coupled systems using common services. The same principles that make SOA desirable in the business domain are also applicable to scientific areas. Grid computing is using a SOA approach for defining core services and providing access to Grid resources [15].

Web services use XML messages that follow the Simple Object Access Protocol (SOAP) standard [48]. A service usually has an attached Web Service Description Language (WSDL) specification which defines a collection of network endpoints—or ports—for the service. Any custom data types are also specified in the WSDL as XML schemas. The SOAP and WSDL specifications are developed by the World Wide Web Consortium (W3C).

These fine-grained Web services can be coupled together to create complex processes using workflows. We will discuss workflow systems further in Section 2.4.

2.2.2 WSRF Services

Standard Web services provide a stateless API to users. This is suitable for many application domains but when operating with physical (and virtual) resources there is a need to store state information. For example, in our client honeypot domain access to VMs would need to be controlled—only one client honeypot can use a VM at once—so state information on VMs is required. Web Service Resource Framework (WSRF) is an extension of Web services that allows the service to store this state information. This is stored in resource variables forming part of the service.

WSRF services match resources to users by generating a unique endpoint reference for each new resource and providing this to users. Subsequent calls to methods that use this resource will include the endpoint reference in an extended SOAP header. WSRF services are defined with a WSDL in the same way as standard Web services are.

2.2.3 Useful Grid Technologies

Here we describe some of the Grid technologies that may be useful to client honeypots:

- **Grid middleware**—is a collection of components enabling the underlying technology that forms a Grid. The components include software services and libraries for resource monitoring, discovery, and management, plus security and file management. Open Grid Services Architecture (OGSA) is developed by the Global Grid Forum (GGF) and describes an architecture for service-oriented Grid computing environments. GT4 (Globus Toolkit 4) [14], gLite [23] and UNICORE [11] are examples of Grid middleware systems.

- **Grid Security Infrastructure (GSI)**—these tools are used for authentication (of users and services), secure communications, and authorisation on the Grid. They are also used for management functions such as overseeing user credentials and group membership information. GSI allows security across organisational boundaries and is therefore not a centrally-managed system. It is a "single sign-on" system for users of the Grid, so handles any delegation of credentials for computations that involve multiple resources. [46]

- **GridFTP**—this is an extension to the standard File Transfer Protocol (FTP) and is defined as part of the Globus Toolkit. The aim is to provide a standard interface for reliable high-performance file transfer over Grid systems. This is a necessity for many Grid applications that need to transfer very large files reliably and securely. The main features are: parallel data transfer, third-party controlled data transfer, GSI security, and fault-recovery. [2]

- **Grid Resource Allocation Manager (GRAM)**—is a standard protocol for communicating with the various job schedulers available, such as Condor, LSF (Load Sharing Facility), and PBS (Portable Batch System). GRAM address security, reliability, and performance concerns, client and server management of resources, and the ability to transfer data to and from remote resources. [12]

- **Storage Resource Broker (SRB)**—is a middleware system providing uniform access to heterogeneous large-scale data storage over the

Grid. Features include logical address spaces, searchable metadata, access control, and collections. SRBs work on top of existing storage systems, real-time data sources, and relational database management systems. Users have access to features such as collaborative sharing, version control, replication, and preservation of distributed data collections. [31]

2.2.4 Grid Service Authoring Tools

Introduce [19] is an open-source toolkit for authoring WSRF compliant Grid services—it aims to reduce the development and deployment effort by abstracting the low-level implementation detail of the Globus Toolkit. Introduce supports the creation of strongly-typed services, where input and output data types to a service are defined and published.

Using a Java based GUI, users define their services and their method signatures with built-in or custom data types. Introduce then generates the low-level service code, leaving method stubs for the user to fill in with code to implement the service functionality. A client application is also generated to test the services. Introduce can then deploy the WSRF services to an application container—GT4, JBoss or Apache Tomcat.

gRAVI (Grid Rapid Application Virtualization Interface) [7] is a plugin for Introduce that helps developers wrap legacy command-line applications with WSRF compliant Grid services. The aim is to require no user code and limited technical knowledge, with the bridging code for the Introduce methods automatically generated. gRAVI can also create a basic Web application to access the service through a Web browser.

2.3 Existing Instrument Middleware

To bring client honeypots to the Grid, we must look at existing work in this field. While there is no other research in this specific area, a client honeypot can be treated as a measurement instrument, so we can look at Grid instrumentation projects. There are several of these which have been developed in the last few years on which to base the integration of instruments with Grid computing. The two most important are the Common Instrument Middleware Architecture (CIMA) [9, 6] and the Grid Enabled Remote Instrumentation with Distributed Control and Computation (GRIDCC) project [24, 17].

2.3.1 CIMA

CIMA provides a framework which has developed implementations from low power wireless sensors through to collaboration and control of large physical sciences instruments. Figure 2.3 shows the CIMA architecture. CIMA allows scientists to access, store and transport data and remotely control scientific instruments and sensor in Grids. The emerging CIMA framework provides the basis to encapsulate client honeypots and develop wide scale Internet measurement systems. CIMA includes integration with GridSphere portals and SRBs. However, the architecture may require extensions in security/privacy, metadata for Internet measurements, enhanced business process execution and data normalisation services.

2.3.2 GRIDCC

GRIDCC provides a similar solution, based on EGEE/gLite middleware [23]. Figure 2.4 shows the GRIDCC architecture. The main components are the Virtual Control Room (VCR) and the Virtual Instrument Grid Service (VIGS). Scientific workflows are executed through the Execution Services (ES) using the Workflow Management Systems (WfMS) to manage Instrument Elements (IE), Compute Elements (CE) and Storage Elements (SE).

GRIDCC has developed a number of instruments based on the EGEE (Enabling Grids for E-sciencE) Grid implementation. The GRIDCC project has implementations that encompass instrumentation on the electrical power

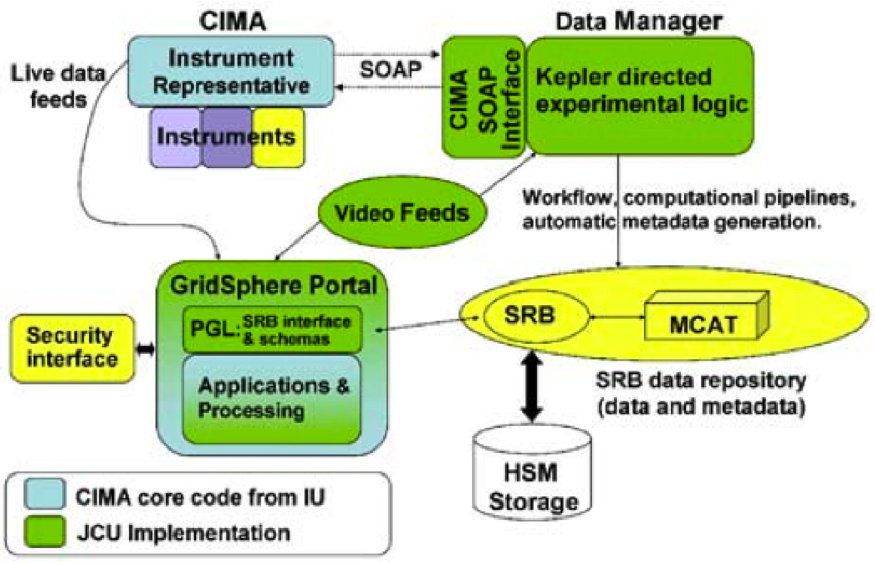

CIMA
Instrument Representative
Instruments
Live data feeds
SOAP
Data Manager
CIMA SOAP Interface
Kepler directed experimental logic
Video Feeds
Workflow, computational pipelines, automatic metadata generation.
GridSphere Portal
PGL: SRB interface & schemas
Applications & Processing
Security interface
SRB
MCAT
SRB data repository (data and metadata)
HSM Storage
CIMA core code from IU
JCU Implementation

Figure 2.3: The CIMA extended architecture [5].

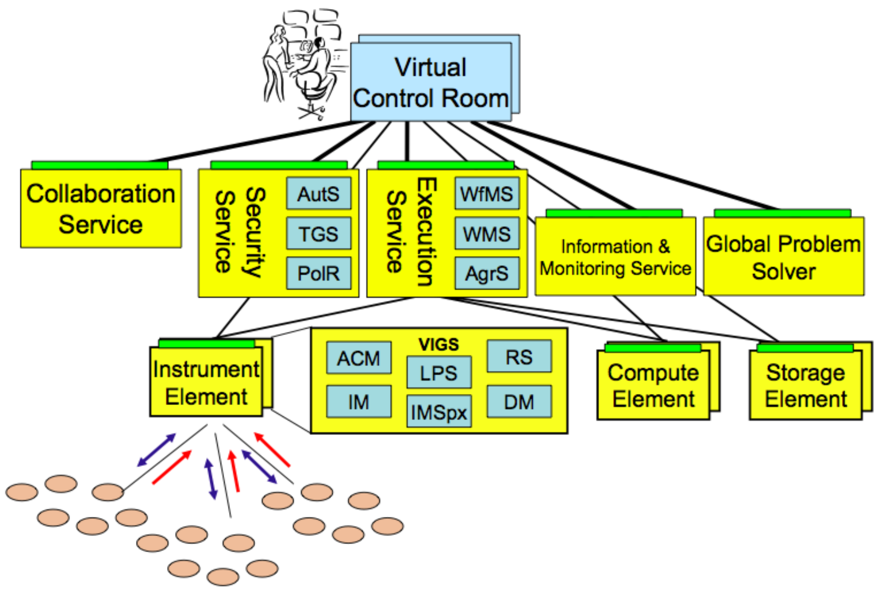


Figure 2.4: GRIDCC architecture [24].

grid, geo-hazard monitoring, meteorology, medical instruments and telecommunications. The GRIDCC project aims to put their Instrument Element middleware into a FPGA (Field-programmable Gate Array) for integration in smaller sensors.

2.4 Workflow Systems

Many business systems use workflows to manage their business processes. Workflows are constructed by connecting multiple tasks according to their dependencies and temporal relationships, which can be represented as either a directed acyclic graph (DAG), or a non-DAG which allows loops in the connections between elements. A DAG-based workflow provides sequence, parallelism, and choice structures. In addition a non-DAG workflows can also include the iteration structure. These four types of workflow structure can be used to construct complex workflows, and sub-workflows can be combined to form a large-scale workflow [49].

Workflows can be represented in a number of ways: by hand (such as documenting a flow chart), writing programs in languages such as Java, or using a workflow language that executes the workflow in a workflow engine. Workflow languages have a number of benefits: a structured representation designed specifically for this purpose, lack of required technical knowledge, and a graphical representation of workflows.

In the recent past, there have been a large number of workflow languages/ systems developed for defining, managing and executing workflows. One recent survey found over 30 scientific workflow systems [39]. It is probable we will soon see consolidation and standardisation as the research area of workflows matures.

Workflow languages for business and scientific domains tend to have differing focuses. Business workflow languages are usually services-oriented with strong support for iteration and control of tasks; this can be considered control-oriented. Scientific workflow languages typically have strong support for pipelining data—modifying some data and passing it on to the next task—and are often DAG-based, these can be considered data-oriented.

In Section 2.3 we introduced two Grid instrument middleware frameworks, CIMA and GRIDCC. Of these two, GRIDCC has provided an open-source implementation for experimentation. The CIMA project uses Kepler workflows [3], a DAG scientific workflow system based on the Ptolemy II system for heterogeneous, concurrent modeling and design. GRIDCC uses the Business Process Execution Language (BPEL) with extensions developed for QoS (Quality of Service) and SLA (Service Level Agreements).

2.4.1 BPEL

BPEL is widely considered the de-facto standard for Web service orchestration, particularly in the business sector, and is more mature than many alternatives. It is a convergence of two languages, WSFL (Web Services Flow Language) and XLANG, developed by IBM and Microsoft respectively. The current version, WS-BPEL version 2.0 [27] was approved by OASIS, the international standards consortium, in April 2007. BPEL is a non-DAG, extensible XML-based language. The specification does not define a graphical notation for visual representation of workflows, so this is left up to individual vendors although flowchart conventions are often used.

Although originally developed for business processes, BPEL has also been suggested for scientific workflows due to its expressibility, standardisation, and maturity of tools [1]. However, BPEL does have some drawbacks for this domain, such as a lack of integration with the Web Service Resource Framework (WSRF) [10].

Grid Integration

A limitation of BPEL is its lack of integration with the Grid. Specifically it does not support WSRF services, large file transfer mechanisms (such as GridFTP) or the GSI security model. However, we have identified extensions to BPEL that aim to integrate this missing Grid functionality.

Dornemann et al [10] have created extensions to BPEL to allow it operate on WSRF (Web Service Resource Framework) resources by introducing a new activity called `gridInvoke` (GI) which is derived from the `invoke` activity and transparently handles the invocation of state-aware WSRF services. GI allows Web services to store the state of operations and properties while still being compatible to standard Web services. Utilising this framework for workflows would prevent a number of possible error situations and assist with integration into the Grid environment.

Amnuaykanjanasin and Nupairoj [4] propose an architecture to enable Grid composition based on OGSI (Open Grid Services Infrastructure) and BPEL. Their approach uses proxy services enabling users to interact with OGSI Grid services, in particular supporting GSI security, factory and notification mechanisms. These proxy services are orchestrated in workflows using standard BPEL and allow run-time bindings. While the approach of using automatically generated proxy services for every Grid service adds complexity, the ability to use Grid security mechanisms, which implies cross-organisational security, is highly beneficial.

A solution to the implementation of large file transfers, is to use Slomiski's proposal of pseudo partners [40] to encapsulate Grid file transfer services. Partners in BPEL are services that a workflow needs and are typically mapped to Web (or Grid) services. However, it is also possible to map BPEL partners to locally implemented services with appropriate WSDL port types provided. Services without a network endpoint are termed pseudo partners as they are not accessible as 'real' Web services but still implement WSDL port types. Tools such as the Web Services Invocation Framework (WSIF) can be used to implement pseudo partners' local services, for example a method in a Java class. Pseudo partners can provide services such as support for large data files transfers using GridFTP, component management XCAT, or interacting with Condor [40].

2.4.2 Taverna

Taverna [28] is an open-source data-oriented scientific workflow system developed as part of the myGrid project. It consists of both a client-side Java GUI application for workflow design and an execution engine for enacting workflows. Like BPEL, Taverna supports the creation of complex workflows containing Web Services and sub-workflows. Individual methods from a Web service are called processors and are connected by data links or explicit control links if data does not flow between processors. Taverna workflows are described using a proprietary language called Scufl (Simple Conceptual Unified Flow language), which is an XML-based language similar to BPEL—but unlike BPEL, Scufl is a data-oriented language in which each processor is passed input data and produces output data for subsequent processors. Data flows are sent in XML form and complex inputs and outputs can be created using a special Taverna processor called an XML splitter that combines or breaks down data streams.

Unlike most data-flow languages, Taverna is a non-DAG system. Iteration, however, is not explicit so does not have the same control has a language like BPEL. Even with just implicit iteration Taverna is still a Turing-compliant language [18]. Implicit iteration is used when a processor is given more inputs than it expects—such as passing a list of elements into a processor that consumes a single element—each input is processed individually until the list has been iterated through and any outputs from the processor are combined into a list once all input items have been processed.

Another kind of Taverna processor is the Java BeanShell. These are pieces of Java code that are interpreted during workflow execution. This allows powerful extensibility to workflows but requires programming knowledge. BeanShells are typically used to transform the output of one processor before it is input into the following processor. Taverna also includes some built-in local processors (which act like BeanShells, with predefined inputs and outputs) for simple tasks such as encoding a byte array into Base64.

2.5 Grid Enabled Internet Instruments

A client honeypot is a type of Internet instrumentation, that is, it takes a measurement of an aspect of the Internet—the number of malicious servers in a sample of URLs. Another type of Internet instrument is the network telescope [26]. These are passive monitoring devices using unused IP address space on a network that capture any packet sent to that address space. The traffic received by network telescopes is called Internet Background Radiation (IBR) [29]. This traffic may be caused by viruses, worms, malicious hackers scanning for vulnerabilities, misconfiguration etc.—because there is no legitimate reason to send traffic to this address space. Successfully detecting and eliminating IBR traffic is important for both reducing the amount of non-productive traffic and also affording a general increase in security.

The goal of the Grid Enabled Internet Instruments (GEII) research project [22] is to develop a framework for using Grid technology combined with emerging Internet instrumentation to provide large scale Internet measurement mechanisms, thus enable new quality measures by which we could determine the state of the Internet. Such measures might include a 'Safety Index' or 'Compromise Index' which indicates the potential for a vulnerable computer to be compromised, a "Malicious Web Server Index" being the percentage or attack capacity of malicious web servers on the Internet. These instruments can capture large amounts of data requiring time consuming analysis so using Grid technologies such as GridFTP, SRBs, and data processing resources would be beneficial. Different components may be spread across organisations to create a Grid of sensors with federated data collection and analysis.

Figure 2.5 shows the proposed architecture of the GEII framework [22]. This shows a number of example instruments (with wrapper envelopes), the central Federated Data Manager, and the Grid services offering processing, storage, security and user interfaces. The example instruments are a VoIP Quality Monitor [21], an IBR Network Telescope [29], and two client honeypots: Capture-HPC (our case study), and HoneyMonkey [45].

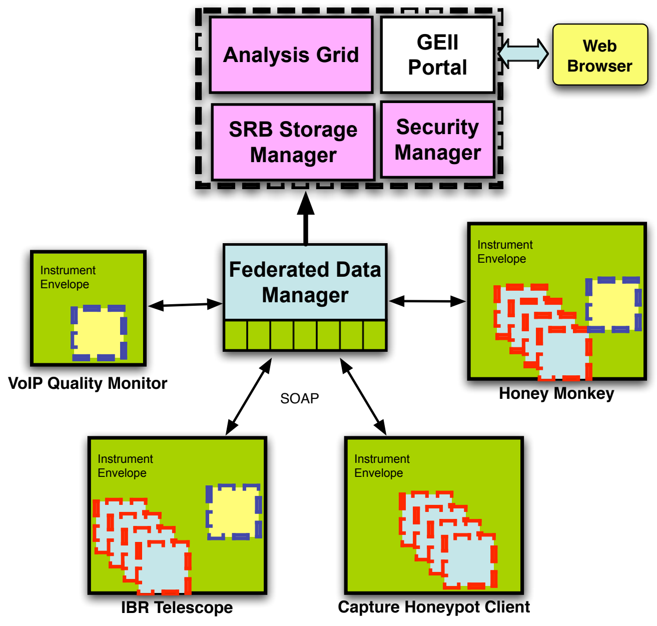


Figure 2.5: High-level view of the GEII architecture [22].

We foresee our research into client honeypot automation being one part of the larger GEII project, with our solutions being extendible to other Internet instruments.

2.6 Review

We now provide a brief review—to tie it all together—of the background and related work covered in this chapter.

We described how client honeypots are used to detect malicious Web servers serving 'drive-by-downloads'—both in their high and low-interaction variations (our case-study Capture-HPC is an example of the former). Grid computing—with its resource sharing across organisations—is a potentially valuable technology for enhancing client honeypot systems. Likewise, workflow languages—orchestrating tasks made up of Web services into complex processes—is another potentially useful technology, with BPEL and Taverna being two suitable languages. Client honeypots are one example of an Internet instrument and we foresee our research being applicable to the wider GEII research project.

We described two instrument middleware projects, CIMA and GRIDCC, which have a similar goal of bringing instruments to the Grid, though the their domain is physical instrumentation. Our focus in this research is to explore workflows so not it is not necessary to use a complete instrument middleware, but future integration is an option.

Chapter 3

Formulation of Requirements

In order to assess the requirements of an automated system we gained necessary experience by manually operating Capture-HPC for moderate scale scans over a period of seven months. This was part of a study of the `.nz` domain, and from this experience we discovered the deficiencies of such a manual approach.

In this chapter we begin by giving an overview of the study and the architecture we used, in Sections 3.1 and 3.2 respectively. Section 3.3 describes the procedure we used including installation, configuration, testing, operation and termination. Section 3.4 looks at the data analysis, presenting some results from the study. In Section 3.5 we formulate general requirements for Grid-based automation, and by creating uses cases, in Section 3.6, more specific requirements . Finally, in Section 3.7 we specify the main requirements to focus on, and define the scope of our research.

3.1 Overview of Study

The study, commissioned by InternetNZ, gathered intelligence on the threat from malicious servers across the `.nz` domain. The aim was to inspect a representative set of the index pages of publicly accessible `.nz` Web servers to determine if they host malicious content. This content may be placed there either deliberately or via a website defacement.

For this study we have completed seven monthly-scans where each scan inspected approximately 250,000 hosts.

For more information on the study, including the project reports, see [20].

3.2 System Architecture

The set-up used for this study consisted of six machines—five Capture-clients and one Capture-server—all running the Fedora Core 7 Linux distribution. The Capture-server is a 2.13GHz Core 2 Duo (2GB memory and 160GB hard disk), while the Capture-clients consist of three 2.4GHz Core 2 Quads (4GB memory and 500GB hard disks) and two 2.8GHz Pentium Ds (1.5GB memory and 80GB hard disks). Figure 3.1 shows the deployment diagram of the set-up. The clients each have VMWare server (and additional VMWare VIX API libraries), with up to three virtual machine instances. Each virtual machine is an image created of Windows XP Service Pack 2 with the CaptureClient and VMWare Tools installed inside. The server has CaptureServer, Squid Web proxy cache, and pdnsd DNS proxy installed. Due to the risk of attacks, security is very important for all the machines so restrictive firewalls were configured and only remote access from static IP addresses is allowed. Key authentication—using a 1024-bit shared key—was a final step used to prevent unauthorised access.

Due to the dynamic nature of the Web and the possible use of IP-tracking—whereby a malicious Web server logs IP addresses and serves benign page when subsequently accessed—we stored the Web pages when we accessed them. This meant malicious content could be later retrieved for analysis [33]. This was implemented using Web and DNS proxies for the clients, configured to persistently cache all data.

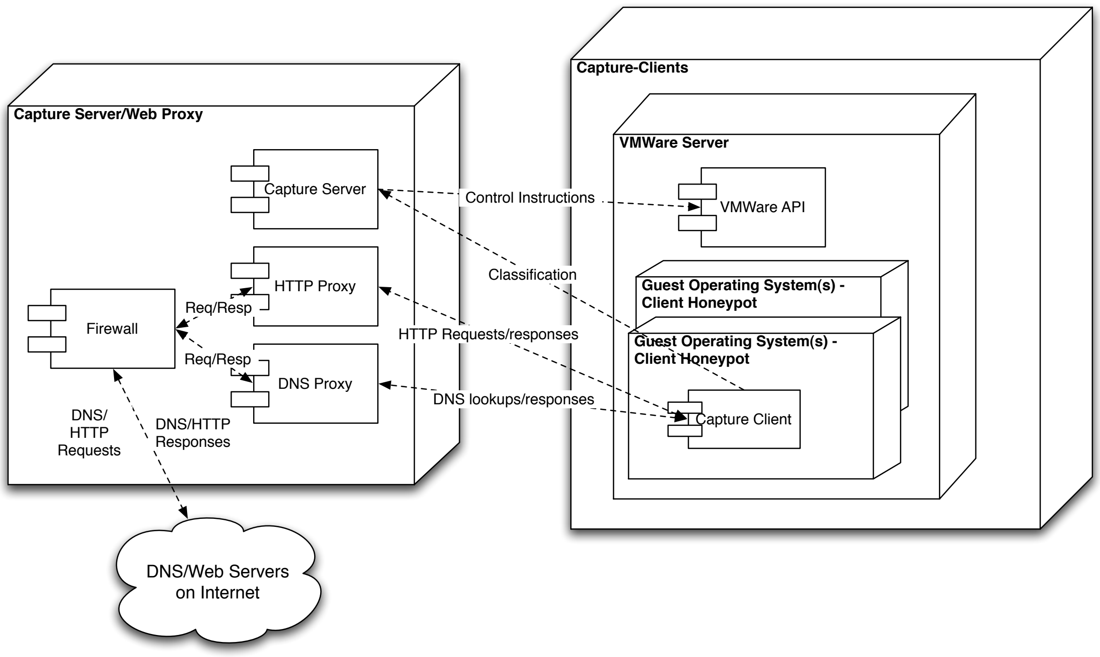


Figure 3.1: Deployment diagram of our Capture-HPC setup.

3.3 Procedure

Running our client honeypot system consisted of five steps: installation, configuration, testing, operation and termination. In each of the following sections we describe the step, problems encountered and improvements to be made.

3.3.1 Installation

All the host machines in the cluster had a default Fedora Core 7 installation. Capture-clients had VMWare Server 1.0.4 and VMWare VIX libraries installed, and a VMWare module compiled for the kernel. The Capture-server had Java 1.6, Squid 2.6, PDNSD 1.2.6 and CaptureServer 2.0 packages installed.

For the client system a new VMWare image was created and Windows XP SP2 was installed, then CaptureClient, VMWare tools and pcap packages were installed. This image was archived and used for future installations.

The process of host machine installation could be simplified if a custom Linux distribution was developed with all the requisite packages included and if remote installation was used.

3.3.2 Configuration

Components such as the firewall, proxies, and CaptureServer needed their configuration files edited and environment variables set. Networking was configured for both the host systems and client systems (the server was assigned a new external IP address for every run to avoid IP tracking). SSH was set up and keys were distributed. The VM image of the client system was allocated to the Capture-clients.

The above configuration was done with a mixture of manual editing (eg. setting up networking parameters such as the host IP address), copying of files (eg. shell file with environment variables), and shell scripting[1] (eg. distributing the VM images). Even tasks that could be scripted were not necessarily simple, it could be an error-prone process because the scripts were fairly unforgiving, and their interpreted and sequential nature meant carrying out a number of steps over a list of Capture-clients could take a long time to execute—up to half an hour.

If the process of installation was done via a custom distribution (as proposed earlier), then some of this configuration could be handled at installation. Also, by giving Capture-servers more control over the set-up of Capture-clients we would reduce the number of scripts and manual edits required.

3.3.3 Testing

Exclusion lists were used by Capture to exclude normal system activity—running processes, registry activity, and file reads/writes. There was a separate exclusion list for each of these three types of activity. The rules are formed by listing the operations and filepaths, with regular expression wildcards allowed. The following is an example of a fileMonitor.exl entry to disregard IE accessing cookies:

```
+ Write C:\\Program Files\\Internet Explorer\\iexplore\.exe
  C:\\Documents and Settings\\.+\\Cookies\\.+
```

Testing was required to check that no erroneous malicious activity was reported during access to normal benign Web pages. This testing had to be manually carried out as it requires knowledge of what normal system activity is and the formulation of rules.

Another important task before operation was to check the optimum number of VMs per Capture-client for maximum throughput. As the number of VMs increased, the usage of resources rose significantly. This was critical during the VM revert operation as the large amount of disk access could

[1]A complete listing of the operational scripts we created is provided in Appendix B.

cause a major bottleneck if multiple reverts were occurring, hence staggering of reverts was necessary. Testing was required to find the "sweet spot" of VMs per machine. Scripts were used to measure throughput (average number of URLs per hour) and configuration files were adjusted as appropriate.

3.3.4 Operation and Termination

Immediately before a run was started the Web and DNS caches were cleared, the list of URLs to test was copied to the Capture-server and its configuration file edited with the parameters for the experiment, eg. visit time and details of the VMs.

A script was then used to start CaptureServer and periodically check the progress, restarting if necessary. Another script was run to give ongoing statistics such as numbers of URLs visited, malicious pages, and time-outs. At completion, the CaptureServer process was terminated and an email notification could optionally be sent.

A complete run of the scan took 5–6 days with our configuration of 13 VMs. During operation this configuration remained static. If a change had been required (such as adding or removing VMs), the CaptureServer would need to have been stopped, configuration file edited, URL list rebuilt, and server restarted. This would cause a drop in performance until the system was properly up and running again. While it was sufficient to have a static configuration for a study of our size, any larger experiment with more machines and over a longer period of time could find this inflexibility a problem; especially because for a very large cluster the probability of at least one hardware failure during the experiment would be quite high.

When undertaking progress checks it was noted that it would be beneficial to be able to select from different levels of information in a progress report. For example, sometimes we would just like to know the percentage of URLs processed and other times we would like to know more in-depth information such as a listing of positive URLs and throughput statistics. Also, the progress statistics only gave overall figures, by not having more fine-grained data a machine running VMWare server could crash and result in reduced overall throughput but would not be immediately obvious. Therefore a management function would be useful.

3.4 Data Analysis

When the operational run was completed all of the CaptureServer log files were aggregated using a script. Pages that were determined to be malicious or had crashed the browser were verified to prevent false positives. These could occur if the exclusion lists had not triggered properly or did not include a benign activity. This verification was done both manually and using a script to filter common occurrences.

Using this list of verified malicious pages the caches were condensed to include only these Web pages. Further runs were then made using a fully patched Windows XP system operating on the cached data. Analysis was also made of the IP addresses of Web servers—which were used to predict their physical locations.

3.4.1 Results from Study

In this section we provide an overview of results from the study, more detailed analysis can be found in [32]. Over the eight months of our study, a total of 291 unique malicious URLs were identified in the `.nz` domain, or about 0.12% of all hosts. Figure 3.2 summarises the results of these monthly scans (no scan was conducted in May).

An average of 73.6 malicious URLs were observed each month. There was a high level of variation, with a standard deviation of 16.2, and fluctuations between 52 (April 2008) and 97 (July 2008) malicious URLs. Regression analysis using least squares determined a p-value of 0.413, meaning there is no evidence of any trend.

Of the malicious URLs identified over the eight month period, a con-

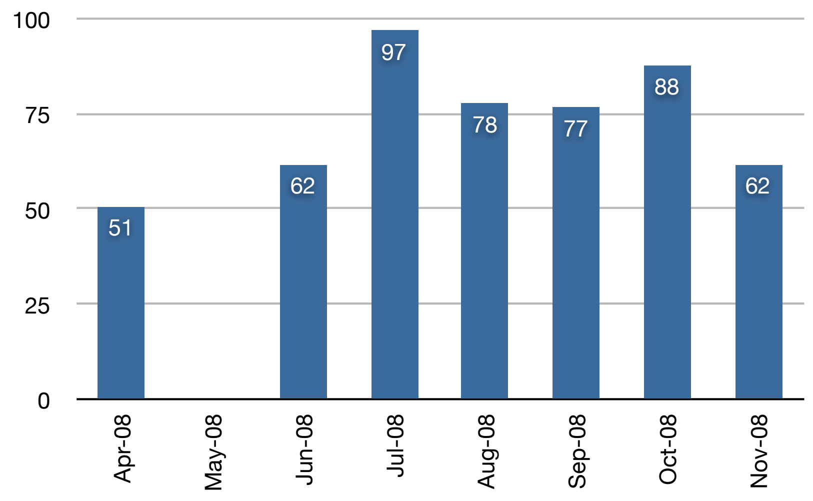


Figure 3.2: Monthly scan results—counts of malicious servers in the `.nz` domain (no scan was conducted in May).

siderable portion of the malicious URLs were newly classified as malicious compared to the previous month. This data is shown in Table 3.1. In July 2008, for instance, nearly 80% of the malicious URLs were newly classified as malicious compared to June 2008. Over the following four months, the percentage of newly classified malicious URLs decreased continuously. In November 2008, about 34% of the malicious URLs were newly classified as malicious compared to October 2008.

All URLs were also inspected with a fully patched system. No URL successfully attacked a system configured with fully patched versions of Windows XP and Internet Explorer.

Table 3.1: Summary of .nz monthly scan results.

Month	**Total malicious**	**New malicious**	**% new malicious**	**Diff. malicious from prev. month**	**% diff. from previous month**
Apr-08	51	-	-	-	-
Jun-08	62	29	46.8	29	46.8
Jul-08	97	75	77.3	77	79.4
Aug-08	78	43	55.1	60	76.9
Sep-08	77	34	44.2	55	71.4
Oct-08	88	38	43.2	47	53.4
Nov-08	62	21	33.9	27	43.5
Average	73.6	40.0	50.1	49.2	61.9

3.5 General Requirements for Grid-based Automation

Using our experience of manually operating a client honeypot system we have formulated the following requirements for automation using a Grid-based architecture:

1. *User-oriented*—the aim being to focus on how the user wants to interact with the instrument. From the user's perspective they have a list of URLs, and want them run on a particular type of client honeypot configuration (OS, browser, visit time, etc.) and are not concerned with the details of the actual set-up. This would require a broker service with look-ups to a dynamic registry of client honeypots. Implementation details, such as the configuration of Capture-clients, can be automated by passing responsibility to the Capture-server. Users should also have options such as the choice of different levels of detail in progress reports.

2. *Loosely-coupled*—interoperability would allow various types of client honeypots to be plugged in with the same interfaces. Instruments could work in conjunction, for example low and high-interaction client honeypots could be used together for faster less-accurate and slower precise runs respectively.

3. *Grid-integrated*—full incorporation of Grid technologies is important. By using Grid services such as SRBs, GridFTP, GSI security, and WSRF services, we can utilise the many Grid resources available, provide a flexible architecture and promote interoperability over multiple sites.

4. *Dynamic*—the system should perform analysis as results are gathered, such real-time analysis could allow for immediate response against new attacks.

5. *Fault-tolerant*—to be successful on the Grid the system should not be too rigid. It needs to be fault-tolerant so Capture-clients are monitored and can be automatically added or removed. Ideally the whole state of the system could be saved and migrated to another system.

For practical reasons we must limit the scope of our research and focus on the core requirements. Therefore, we will focus on the first three—a system that is user-oriented, loosely-coupled and Grid-integrated.

3.6 Use Cases

We have utilised use cases to help gather key requirements and help define the role the system must take in user interactions. Figure 3.3 shows a use case diagram modelling system functionality from the user's perspective. The simplest non-trivial use case scenario would be a user submitting a list of URLs to a client honeypot and simply receiving the results. In section

3.6.1 we expand upon that scenario with a more complex experiment, and in Section 3.6.2 we provide some variations to that specific scenario.

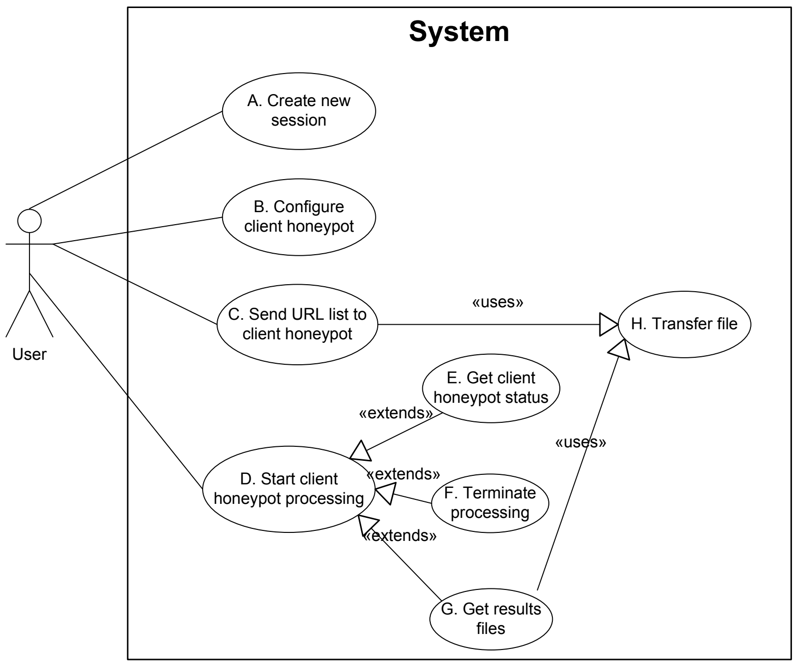


Figure 3.3: Use case diagram showing the main functionality of the system from the User actor's perspective.

3.6.1 Example Scenario

In this specification we describe a particular path through a use case written from the actor's point of view, listing the steps needed to accomplish the goal.

Use case name: Testing URLs on two differently configured systems.

Overview: The actor in this use case is a client honeypot user, eg. a security researcher. The goal is to test whether a list of malicious URLs can exploit vulnerabilities on two differently configured systems. Each system is a virtual machine and we assume images exist for each configuration. To achieve this goal, a URL list is batch processed on the first VM, and then any malicious URLs are feed into the second VM for further testing. An example of this would be testing on a non-patched system and subsequently testing on a patched system—by passing only the malicious URLs to the second VM we significantly speed up the process.

Main scenario: Below shows the steps that make up this scenario. The sub-use cases from Figure 3.3 are referenced in brackets.

1. User identifies a registered client honeypot system and logs in (A).
2. System authenticates the user (A).
3. User passes configuration options for first system (B).
4. User transfers a list of URLs to be tested (C, H).
5. User instructs the system to begin processing (D).
6. System invokes the client honeypot to begin processing on the first VM (D).
7. User optionally requests the status of processing (E).
8. System outputs summary information of client honeypot progress (G).
9. When processing is finished, system informs user (C, D).
10. System transfers completed log files (G, H).
11. User aggregates the log files, extracting the malicious URLs into a new list (outside the system).
12. User passes configuration options for second system (B).
13. User transfers their new list of aggregated malicious URLs (C, H).
14. User instructs the system to begin processing (D).

15. System invokes the client honeypot to begin processing on the second VM (D).

16. User optionally requests the status of processing (E).

17. System outputs summary information of client honeypot progress (G).

18. When processing is finished, system informs user (C, D).

19. System transfers completed log files (G, H).

20. User aggregates the log files, creating results lists— a list of URLs that are malicious on both systems and another for those that are malicious only on the first (outside the system).

Exceptions:
11a. There are zero malicious URLs—user skips processing URLs on the second VM and goes to step 20 (creating the results lists).

3.6.2 Scenario Variations

Other client honeypot experiment use cases include:

- *Combining two different client honeypot systems*—in this scenario a user passes URLs to one client honeypot system and feeds the output into another system in the same organisation. An example would be firstly using a low-interaction client honeypot and then feeding the malicious pages into a high-interaction client honeypot, thus utilising the speed of the former with the accuracy of the latter. The main difference to the scenario specified in Section 3.6.1 is the need for the user to identify and log into the second client honeypot.

- *Combining client honeypots from multiple domains*—the goal of this use case is to test URLs on client honeypots located in different domains. An example would be two universities collaborating—sharing their client honeypot resources. Compared to the scenario in Section 3.6.1 the security requirements will be higher as virtual organisation security systems such as GSI would be used. Also, there may be requirements to anonymise data to meet organisational requirements.

3.7 Main Requirements and Scope of Research

In this chapter we have uncovered a number of requirements for automating client honeypot systems. For practical reasons we must limit the scope of our research and focus on the core requirements—the goal is to implement a prototype system which allows further functionality to be added later.

In Section 3.5 we specified the three general requirements we would focus on a system that is user-oriented, loosely-coupled and Grid-integrated. The detailed requirements are specified in terms of use cases, given in Section 3.6.

We define the scope of this research as follows:

- Focus on creating WSRF Grid services and the practicalities of workflows, such as combining multiple VM images and heterogeneous systems.
- Focus on the operation and data analysis phases of the client honeypot lifecycle. As already mentioned, the installation and configuration phases could be largely automated by using a custom Linux distribution.
- Limit to a single organisation. Therefore security issues, resource management, and virtual organisations are not covered.
- Not include fault tolerance at this stage. Workflows and services will throw generic exceptions on error conditions, but we will not focus on fault recovery.

Chapter 4

Design and Implementation

This chapter focuses on the design and implementation of our system, created to meet the requirements we gathered in the previous chapter. Section 4.1 describes our system architecture. Section 4.2 investigates the workflow design including: areas of application, an evaluation of BPEL and Taverna, and sample client honeypot workflows. Section 4.3 looks at our Grid services that form the ClientHoneypotManager and ClientHoneypotWrapper components. Section 4.4 describes the implementation of our prototype system, which includes: WSRF services, a Web portal and Taverna workflows. We conclude the chapter, in Section 4.5, with a summary of the material covered.

4.1 Architecture

To meet the requirements we identified in Section 3.7—a user-oriented, loosely-coupled and Grid-integrated system—we needed to expand the client honeypot architecture. Components were required that allow integration of existing client honeypots, manage and provide access to these systems, and interface with Grid middleware.

Our design uses a service-oriented architecture (SOA) approach; this provides a necessary level of abstraction and interoperability using common services. Above these services we use workflows to manage the orchestration of services into larger processes, automating the complex operation of client honeypot systems. A workflow system allows processes to be made into components and should enable users of the system to create and modify these complex processes with ideally little technical knowledge.

Figure 4.1 shows our system architecture. An existing client honeypot is encapsulated with the ClientHoneypotWrapper services. Above this is the ClientHoneypotManager component which acts as a broker between users and registered client honeypots. It also interfaces with Grid services such as job schedulers and SRBs. User access to the ClientHoneypotManager and the workflow system is provided through a Web portal.

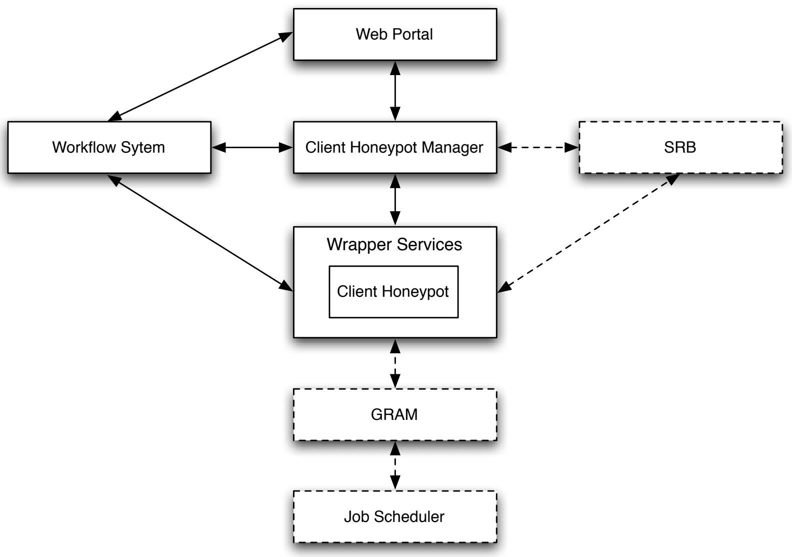


Figure 4.1: Architecture diagram of our system design—items with dashed lines are existing Grid components.

The ClientHoneypotWrapper WSRF services provide a standard interface for all client honeypot instances. An implementation of this service can either call an exposed API in the client honeypot or provide a bridge to the application directly. The service can be deployed as a GRAM (Grid Resource

Allocation Manager) job for submission to a job scheduler.

The ClientHoneypotManager will use Grid indexing services for publication and subscription of registered ClientHoneypotWrappers. Data published could include the capabilities of the system, with specification of VM configurations. Via the Web portal, users could then create a client honeypot session by specifying only the URLs to test and any requirements for the system to test on, letting the ClientHoneypotManager deal with the job scheduling and the details of the client honeypot operation.

The orchestration of these WSRF services can be achieved either with a custom application, requiring pre-compiled code and users to have programming knowledge; or using workflow systems, graphically modelling the operation and potentially allowing non-technical users to create complex processes. Initially we investigated how workflow systems could benefit our system.

4.2 Workflow Design

4.2.1 Areas of Application

We saw workflows as an important mechanisms for automating the various processes used in client honeypot management. There are two main areas where workflows are potentially beneficial: (1) operating a specific client honeypot and (2) federation of multiple client honeypots. These are detailed in the following two sections.

Operating a Client Honeypot

Each client honeypot implementation has its own internal processes. In a high-interaction client honeypot the main process involves the server component delegating URLs to client components, analysing any state changes, then classifying the results. If the individual tasks that make up this process where exposed as Web Services, the entire process could be modelled with workflows. By using workflows the processes can be viewed graphically and modified by someone without technical expertise of that client honeypot.

An example of a potential modification to a process made via a change to the workflow is the implementation of a divide-and-conquer algorithm [38]. This algorithm visits 20 URLs at once (in separate windows), and determines if there is a malicious URL amongst them, if there is, the set will be divided and re-tested until the malicious page or pages are determined. Due to the low count of malicious pages making division rare, the parallel visiting of pages increases throughput significantly. Such a change in the operation process could be made by creating a new workflow and graphically inserting the appropriate control structure such as `while` loops and `if` statements. The previous operational behaviour would still be accessible through the original workflow. This contrasts with the current approach where changes require the server component to be recompiled and redeployed.

Tight control over the internal processes could also allow runtime modifications of the system through Web services. An example would be the ClientHoneypotManager adding or subtracting CaptureClients as resources' availability changed.

While fine-grained control of client honeypot systems is useful, it brings a few disadvantages too. There will be a reasonable amount of work involved in modifying any client honeypots to expose its internal tasks as Web services. There will be a performance penalty due to the added communication and overhead of SOAP messages used in Web services. Also, it may be difficult to create standard interfaces so that workflows created for one client honeypot are compatible with another client honeypot system. This is because each system's internal implementations is unique, data types and methods vary, so code may need to be rewritten to enable convergence.

Federation of Client Honeypots

This approach treats a client honeypot as a black box to which we pass in URLs and it passes out classification information, thus we are not concerned with the internal workings. As well as the classification of a URL we can also gather useful additional meta data on the Web page itself, ie. popularity of site (from ranking sites), geographic location of the Web server, number of redirects, etc. Using this meta data, machine learning techniques can be implemented to generate better profiles of malicious pages, eg. for the formulation of signatures for low-interaction client honeypots [34].

Workflows can be used that filter URLs returned from one client honeypot using additional meta data, and pass matching URLs to another client honeypot to inspect. An example could be a workflow that tested if client honeypot location—in terms of country domain—affected whether the Web page was delivered as malicious. The workflow might test an initial list of URLs on a New Zealand honeypot, filter the resulting malicious URLs to only those of high popularity, pass these URLs to a client honeypot located the UK, and finally aggregate the results.

This approach to workflow usage would require a standard WSRF service interface to be developed which can be applied to any client honeypot instance. A specific client honeypot would then be wrapped with this generic interface, requiring little or no modification to its existing code. The post-processing of pages for additional meta data could be an extensible part of the service.

4.2.2 BPEL and Taverna Evaluation

In Sections 2.4.1 and 2.4.2 we gave an overview of the BPEL and Taverna workflow systems. We now provide an evaluation of these systems for suitability to model client honeypot workflows.

BPEL is a control-oriented workflow language. Its main strengths are its maturity and wide support by industry tools and engines. Being a non-DAG language it supports iteration (which is explicit), allowing complex control structures to be expressed. A potential imitation of BPEL is it only supports WSRF through an extension, but the extension is not compatible with the current release (which has significant improvements in concurrency). Another drawback is that the creation of workflows is a complicated process so is not suited to non-technical users. As one of the motivations we gave for using workflows was to allow these non-techinical users to create and modify client honeypot processes, this is significant.

Taverna is a data-oriented workflow language. Its strengths are: a simpler system to develop workflows in, so non technical users should be able to use it; and the integration of WSRF services is an example of the focus on Grid computing. GSI security integration is also in development. Potential limitations include the lack of explicit iteration affecting workflow design, and the system's immaturity—meaning there are bugs and functionality lacking in some areas.

The two approaches to workflow usage we identified in the previous section—operating a specific client honeypot and federation of multiple client honeypots—have different requirements for the workflow system they are implemented in. We now look at how these requirements apply to BPEL and Taverna.

The control-oriented operation of a specific client honeypot fits the BPEL workflow design, particularly the ability to include iteration. The increased complexity of BPEL workflow design is not a significant issue because it would likely be users with technical knowledge that would modify internal honeypot processes.

The federation of multiple client honeypots is more of a data-oriented process so suits Taverna's model better. The simpler design process of Taverna will benefit non-technical users, and the integration to the Grid is more important at this higher data-processing level than it is at the lower control-oriented level.

In the next two sections we will begin modelling these two workflow approaches with the counterpart workflow languages we have identified—BPEL and Taverna.

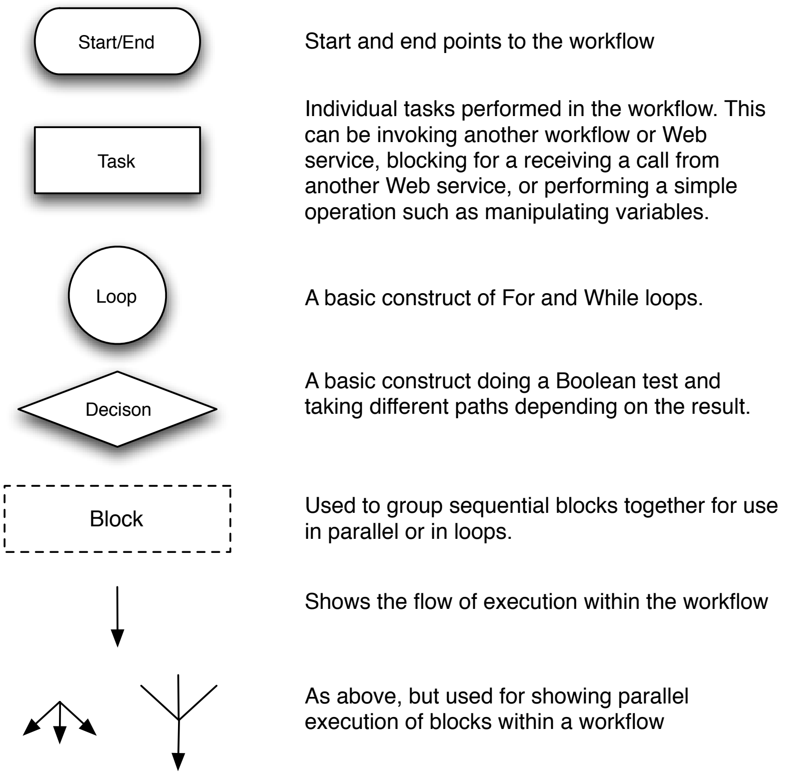
4.2.3 BPEL Workflows for Client Honeypot Modelling

To assess the expressability of BPEL as a workflow language for operating a client honeypot we constructed workflows to model the operation of our case study, Capture-HPC. We applied the constructs and semantics of WS-BPEL version 2.0, currently the latest version. Of interest, this version introduces new activity types of `<repeatUntil>`, and `<forEach>` (with both parallel and sequential versions).

We now describe some of the issues encountered using BPEL and present example workflows that try to address these identified issues. The functionality of Capture-HPC is exposed as Web services and the workflows invoke and receive calls from these Web services. For ease of communication we have chosen to model our workflows graphically, however there is no standard graphical notation for BPEL. Most vendors have invented their own notation or use the BPMN (Business Process Modelling Notation). These notations all use a flowchart model—such as that used in UML activity diagrams—with the addition of a block structure. At this stage we have modelled our workflows in a simple graphical notation, Figure 4.2 shows a key for the different constructs used. At a later stage, when a workflow engine and tools have been selected, we can map to the appropriate notation used.

An issue that we encountered when using BPEL was that the specification does not define what happens if a workflow blocking for a `<receive>` call, receives more than one call to the Web service. Therefore we cannot use a single `<receive>` to deal with all clients returning results to the CaptureServer in the Captue-HPC system. To avoid this we could implement many top level workflows as a single BPEL engine can handle hundreds of concurrent instances.

Figure 4.3 shows a workflow modelling the operation of Capture-HPC using the `<forEach>` construct. The CaptureServer is initially passed a list of URLs. For each registered CaptureClient a while loop is run in parallel. This passes a URL to the CaptureClient to visit, receives state changes, and determines if the Web server is malicious—in which case it resets the



Start and end points to the workflow

Individual tasks performed in the workflow. This can be invoking another workflow or Web service, blocking for a receiving a call from another Web service, or performing a simple operation such as manipulating variables.

A basic construct of For and While loops.

A basic construct doing a Boolean test and taking different paths depending on the result.

Used to group sequential blocks together for use in parallel or in loops.

Shows the flow of execution within the workflow

As above, but used for showing parallel execution of blocks within a workflow

Figure 4.2: Workflow diagrams' key.

VM that the CaptureClient is running inside. The network data captured is transferred to a storage location and the URL classification list is updated. These loops continue until there are no more URLs left to test, whereby the classification list is returned. By having a separate `<receive>` for each CaptureClient in its parallel loop we avoid the issue of multiple calls to a single `<receive>`.

The `<forEach>` construct allows for the execution of a sequential block an arbitrary number of times in parallel. The number of concurrent instances of this block can be determined at run time before actual execution, but once it has begun there is no way to add another concurrent instance. To fully control a honeypot's operation we need to delegate URLs to clients in parallel, and allow the number of these connected clients to increase or decrease during operation.

Figure 4.4 shows an example workflow that avoids this concurrency limitation. Initially CaptureServer receives a list of URLs. Then two processes are run in parallel—the first delegates URLs to free CaptureClients until there are no more left to delegate, the second performs routine checks for timeout conditions on assigned URLs until all tasks are completed. Both update status lists to keep track of CaptureClients and URLs. When both parallel branches have completed the findings are returned.

An event handler is used to receive the results from CaptureClients. The specification states that a business process can receive event handler messages concurrently with the normal activity of the scope to which the event handler is attached, thus allowing events to occur at arbitrary times while the normal scope is still active [27]. Using this design clients can be added and removed by modifying the status lists, thus avoiding the fixed concurrency issue of the `<forEach>` construct.

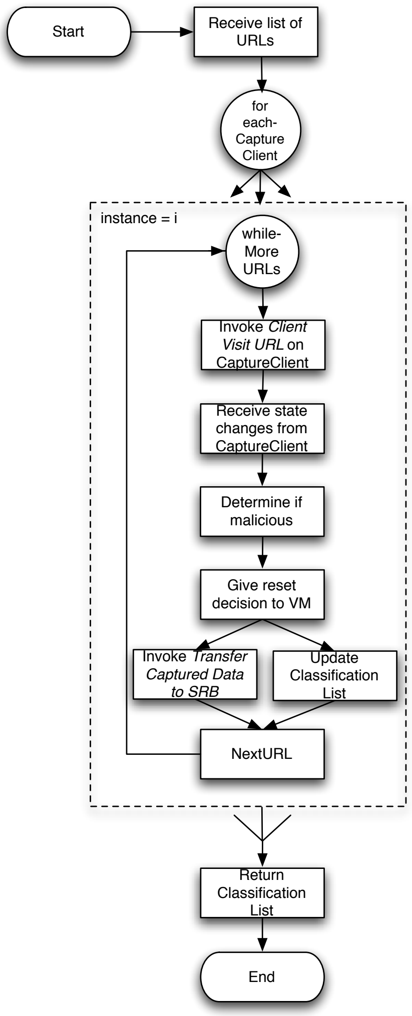

Start
Receive list of URLs
for each-Capture Client
instance = i
while-More URLs
Invoke *Client Visit URL* on CaptureClient
Receive state changes from CaptureClient
Determine if malicious
Give reset decision to VM
Invoke *Transfer Captured Data to SRB*
Update Classification List
NextURL
Return Classification List
End

Figure 4.3: BPEL workflow of CaptureServer's operation with parallel forEach construct. Parallel while loops are run for each registered CaptureClient, delegating URLs and processing results.

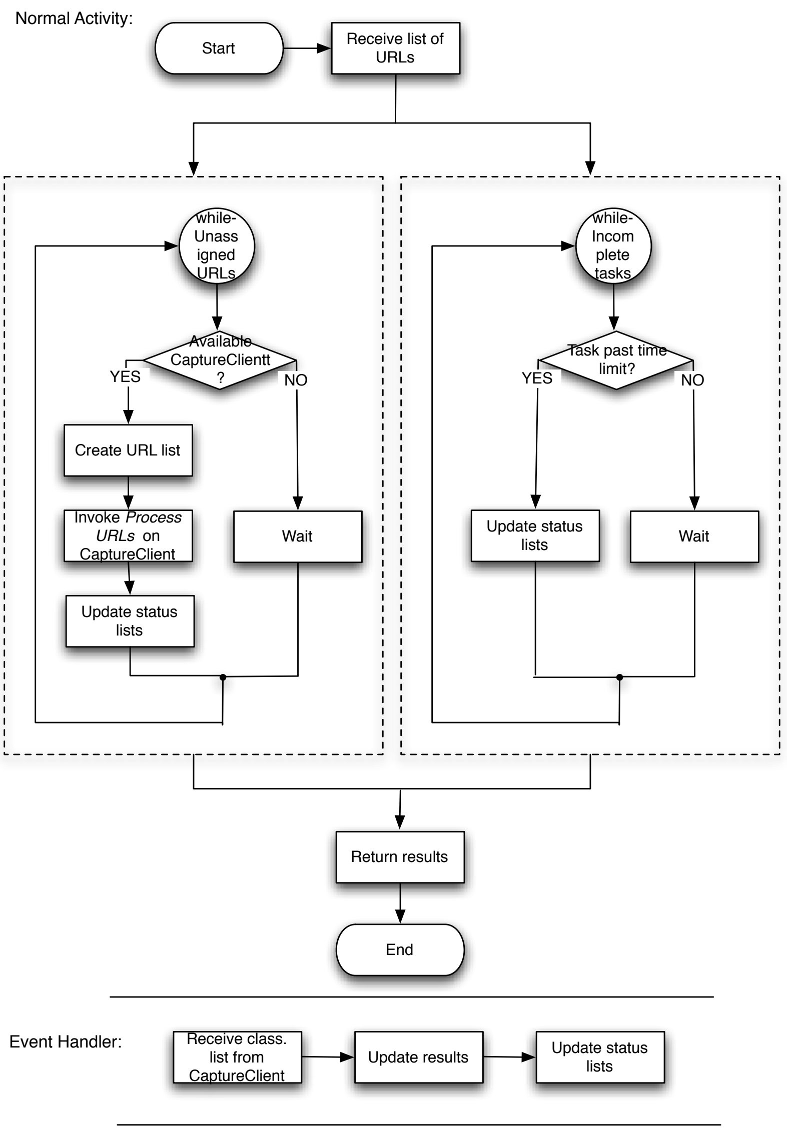


Figure 4.4: BPEL workflow modelling CaptureServer delegating URLs to CaptureClients. Parallel processes delegate URLs and check timeout conditions, while an event handler processes results.

4.2.4 Taverna Workflows for Client Honeypot Federation

In this section we use Taverna to model data-oriented workflows operating on wrapper WSRF services that will encapsulate specific client honeypots. In the same way we did with the previous BPEL workflows, we will again use a simple graphical notation (Figure 4.2 specifies the key). For simplicity, some of the non-essential data flow lines are omitted.

Figure 4.5 shows a simple workflow that will process a URL list and return the output files (URL classification lists and log files). The inputs to the workflow are the URL list and optionally configuration options for the client honeypot. A resource key is created which will identify this particular client honeypot session. Then the `Process URL List` method is called, passing the resource key, URL files and any configuration options. This is a blocking method so while this is operating the `Get Status` method outputs the progress. When the processing is complete the workflow will move on to get the files. The method to get the file list will return multiple file names, using Taverna's implicit iteration, each of these files will be individually retrieved in the `Get File` method.

Figure 4.6 shows a more complex workflow—processing URLs on two differently configured systems—which is the use case we described in Section 3.6.1. The workflow tests whether a list of malicious URLs can exploit vulnerabilities on two differently configured systems. So a URL list is processed on the first VM, then any malicious URLs are fed into the second VM for further testing. The first section is the same as our previous process in Figure 4.5, the difference is the output is parsed (to create a new input URL list from the output's malicious list) and this list is input into the second `Process URL List` call (also passing any configuration options). When the output files are retrieved from this client honeypot session the results are aggregated to provide classification lists for the URLs over the two systems.

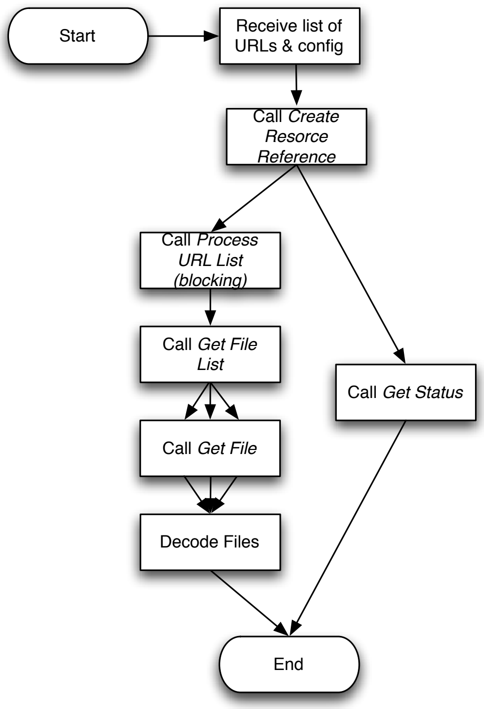

Start
Receive list of URLs & config
Call *Create Resorce Reference*
Call *Process URL List (blocking)*
Call *Get File List*
Call *Get File*
Decode Files
Call *Get Status*
End

Figure 4.5: Taverna workflow modelling basic operation of a client honeypot.

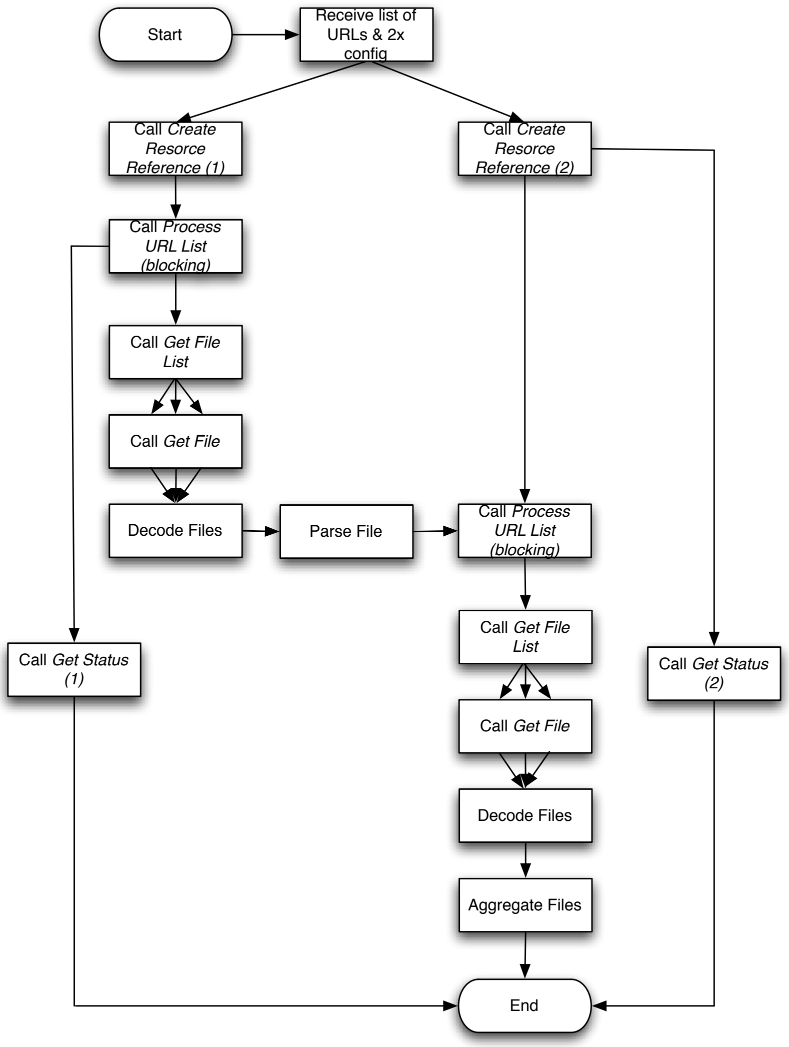

Start
Receive list of URLs & 2x config
Call *Create Resorce Reference (1)*
Call *Create Resorce Reference (2)*
Call *Process URL List (blocking)*
Call *Get File List*
Call *Get File*
Decode Files
Parse File
Call *Process URL List (blocking)*
Call *Get Status (1)*
Call *Get File List*
Call *Get Status (2)*
Call *Get File*
Decode Files
Aggregate Files
End

Figure 4.6: Taverna workflow modelling processing URLs on two differently configured systems.

4.3 Grid Service Design

In this section we describe the WSRF interfaces designed for the two main components of our system—the ClientHoneypotManager and the ClientHoneypotWrapper services. Not all details are provided—for example custom data types—these are provided in Appendix C. In section 4.3.3 we present an interaction diagram that shows these services and their methods in operation.

4.3.1 ClientHoneypotManager

The purpose of this service is to be a broker between users and a number of registered client honeypots. The user initially creates a new session by calling the *initiateSession* method, this returns a reference to the *CHSessionManager* service context. The *registerClientHoneypot* and *deregisterClientHoneypot* methods are used to add or remove a client honeypot's information from the indexing service. The methods are defined as:

- CHSessionManagerReference initiateSession ();
- void registerClientHoneypot (ClientHoneypotDescriptor info);
- void deregisterClientHoneypot (ClientHoneypotDescriptor info);

The *CHSessionManager* service context deals with a specific user's session. It stores a Session resource (see section below for complex type definitions). It has methods *processURLs* and *terminateProcessing* for starting or stopping processing a list of URLs, and *getStatus* to get info on the current session. These methods are defined as:

- String processURLs (int priority, ClientHoneypotConfig settings, Base64Binary urlListFile, Reference endpointRef);
- void terminateProcessing ();
- SessionStatus getStatus ();

4.3.2 ClientHoneypotWrapper

This service is designed to provide a WSRF wrapper for client honeypots. The aim is to use a standardised generic client honeypot interface so that the ClientHoneypotManager has a single service type to interact with. Each client honeypot system (ie. Capture-HPC, HoneyMonkey, etc.) will require a different implementation of this interface, but the service will be designed so that these specific implementations require minimal changes at the back-end to interact with the honeypots they are wrapping.

We have based this service on the services that the authoring tool gRAVI creates to wrap legacy applications (described in Section 2.2.4). This ability to wrap command-line applications with little code modification fits with what we require. To implement these services we will have to standardise the parameters and data types. Additionally we may need to extend the gRAVI services to integrate resource allocation.

If we were to write these services by hand—without gRAVI—we would have to write: low level Globus Toolkit code such as configuring resources, maintenance code such as updating the service context's state, code for handling the various error conditions arising in the services, and client code for testing and debugging.

The main service has the operation *createResultsResource* which returns a new reference which is used to refer to this particular session. The *runClientHoneypotSh* method will start the client honeypot with the provided arguments and files (such as URL lists and configuration files). The resRef parameter is optional, if null is provided a new reference is created and this value is returned.

- ResourceReference createResultsResource ();
- ResourceReference runClientHoneypotSh (ResourceReference resRef, String[] arguments, String[] filenames, Base64Binary[] inputFiles);

The service context for the client honeypot session has a number of methods for transferring files, requesting the status, and terminating processing.

These are defined as:

- String[] getFileList();
- Base64Binary getSingleFileSOAP(String filename);
- Status getStatus();
- boolean killProcess();
- boolean sendFileSOAP(String filename, Base64Binary inputFile);

The methods provided for transfering files via SOAP also have GridFTP versions (see Appendix C).

4.3.3 Interaction Diagram

Figure 4.7 shows a sequence diagram modelling the sequential operation of the services we have specified in the previous two sections. This shows the process of initiating a session and the user submitting a URL-list to be processed. The manager looks up a suitable client honeypot, creates a resource and starts processing. Status is periodically checked and a call-back is provided to the user. When processing is completed the files are transferred and the manager returns the location of these files to the user. The status can also be checked directly from the user, separate from the periodic status updates. The User component could either be a workflow, Web portal, or other application; while the Capture-HPC class could be replaced with any specific client honeypot.

4.4 System Implementation

In this section we describe the proof-of-concept implementation of our system design. Figure 4.8 shows the components of our implemented system.

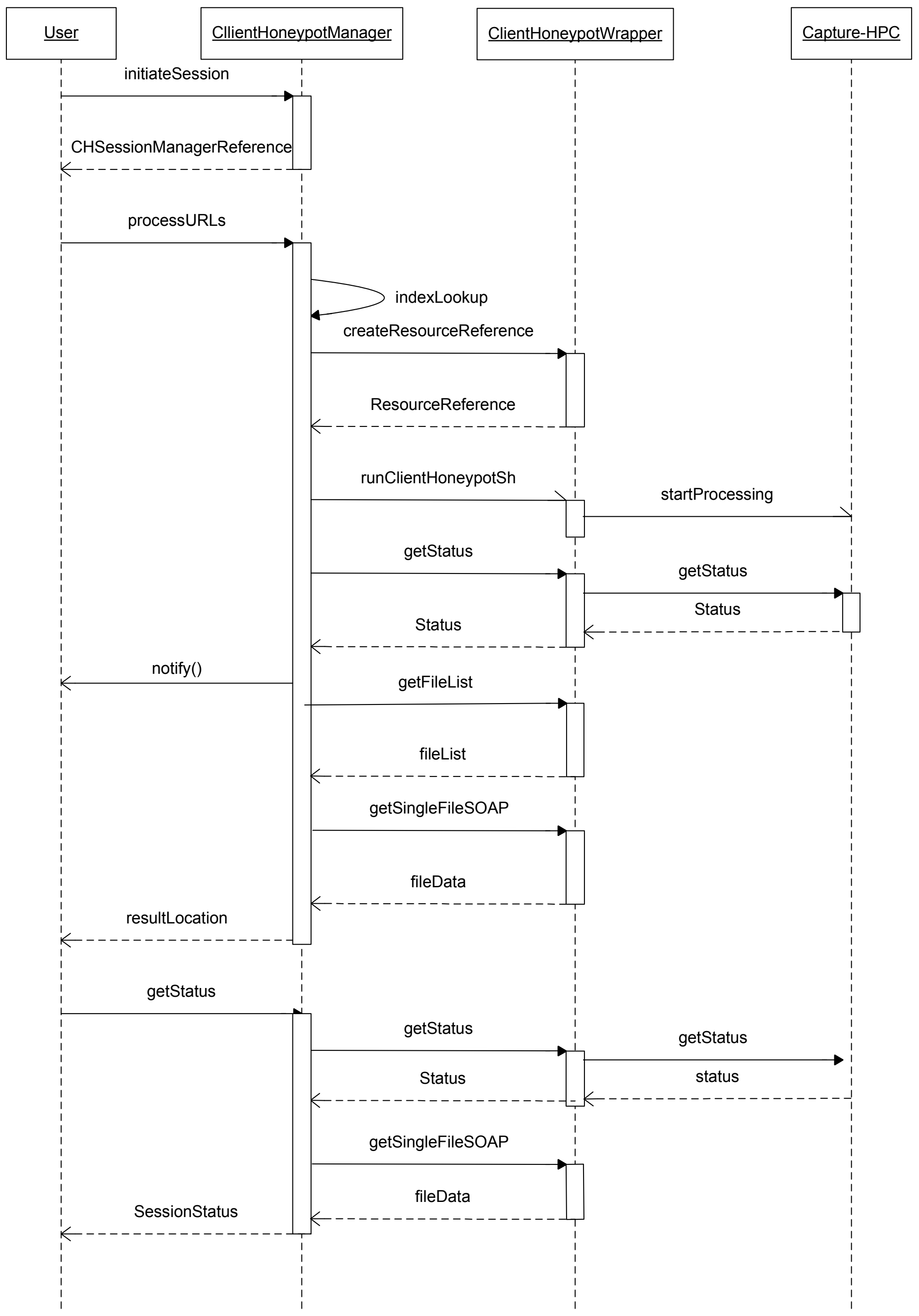


Figure 4.7: Sequence diagram showing the interaction of components in our system when processing a list of URLs.

At the bottom of the diagram we have have an unmodified client honeypot—in our initial implementation this is Capture-HPC. Above this is a custom shell script which provides a bridge between the generalisation of the wrapper services and specification of our specific client honeypot—Capture-HPC. Above the custom shell script is the ClientHoneypotWrapper WSRF service which is a standardised component, requiring only the filepath to the custom script to be modified and a unique service URL given when deployed. The wrapper service and script implementations are described further in Section 4.4.1. Above this is the Web portal (described in Section 4.4.2) and the Taverna workflow engine and user interface components (described in Section 4.4.3). We use these workflow components to design and enact workflows that operate on our WSRF wrapper services.

Each of the described components can be located on different physical machines, allowing flexibility in deployment. The Taverna workflow system can be considered one part of the ClientHoneypotManger component, the rest has been omitted to save time.

4.4.1 Custom Script and Wrapper Services

The custom Unix shell script converts the standardised parameters and data types of the WSRF service to the specifics of Capture-HPC. The script also adds functionality not in CaptureServer such as terminating processing and outputting progress statistics. The usage of this intermediate script means there is no need to modify either the existing client honeypot system or the WSRF services. Developers of other client honeypots could simply write their own 'wrapper' script to also use our WSRF services.

An alternative with tighter integration is to modify the WSRF services to call an API that the client honeypot exposes. Capture-HPC 2.5 does not currently expose an API.

The WSRF Grid services we use are the core services created by gRAVI. These can be deployed to either a Globus container or an Apache Tomcat installation that has the GT4 libraries installed. They also have the ability

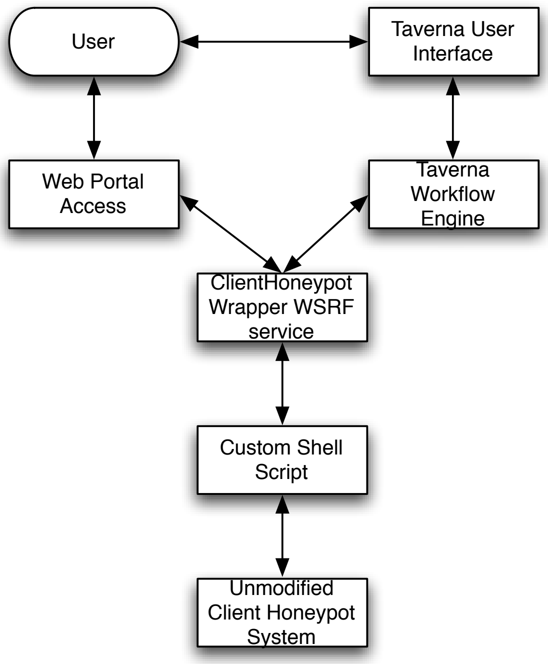


Figure 4.8: Component diagram of our prototype system.

to be deployed as a GRAM job but we have not integrated it with any job schedulers. This means that currently there is no resource allocation mechanism to limit to single user access of high-interaction client honeypots' resources. This would be necessary for most high-interaction client honeypots in a production environment, as two instances could not use the same virtual machines simultaneously.

In our tests of the services, options for security in the container were not enabled.

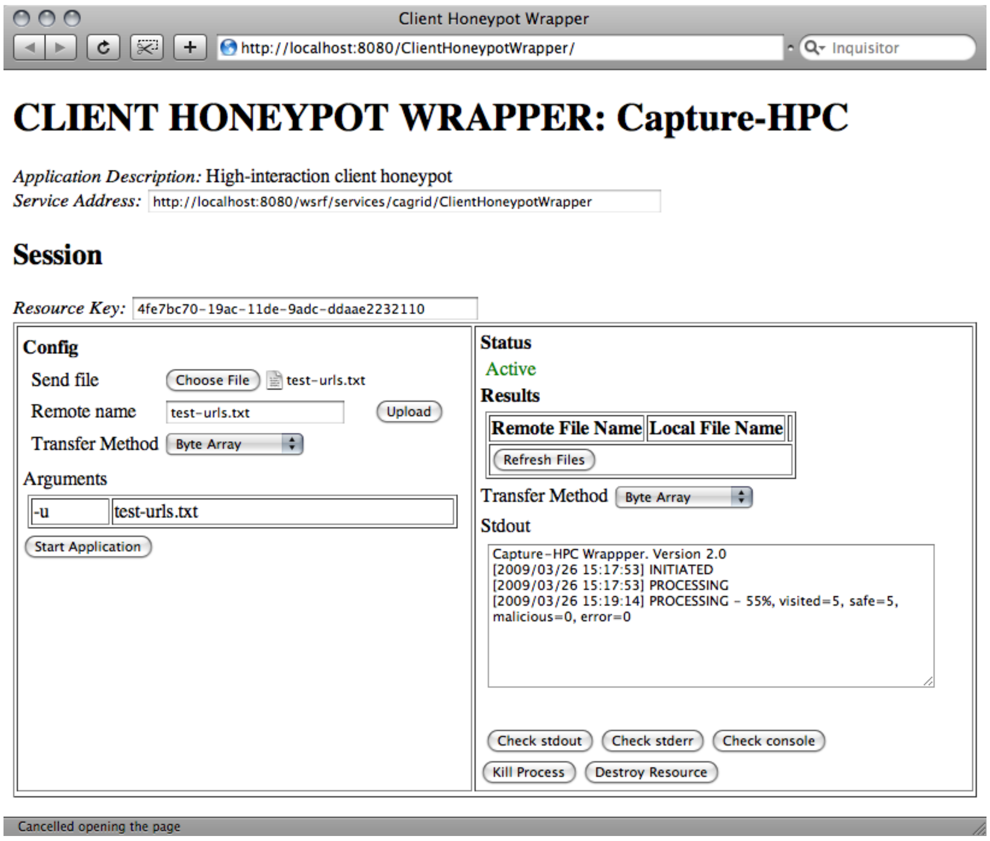
4.4.2 Web Portal

The Web portal component of our system allows access to our ClientHoneypotWrapper service through a Web browser. This basic Web application is a modification of the one provided in gRAVI, Figure 4.9 shows a screenshot. The user can upload URL lists and optional configuration files, and pass standardised operational parameters to the service. After the user starts the client honeypot, progress stats are displayed until processing is complete whereby they can download the generated classifications. In our prototype system the Web portal is not integrated with our Taverna workflows, though it does allow the progress stats of workflows to be viewed if the resource key from the workflow is imported.

4.4.3 Taverna Workflows

In this section we describe our experiences of using Taverna to implement the workflows we designed in Section 4.2.4. There were a number of problems we encountered, the main issues were:

- A GUI bug in Taverna 2.0 prevented us scavenging the WSDL for our services. When the WSDLs for the main service and the context service are both added, the service's methods were merged with around half the method names not displayed, meaning it was impossible to add



CLIENT HONEYPOT WRAPPER: Capture-HPC

Application Description: High-interaction client honeypot
Service Address: http://localhost:8080/wsrf/services/cagrid/ClientHoneypotWrapper

Session

Resource Key: 4fe7bc70-19ac-11de-9adc-ddaae2232110

Config
Send file: test-urls.txt
Remote name: test-urls.txt
Transfer Method: Byte Array
Arguments: -u test-urls.txt

Status
Active
Results

Remote File Name	Local File Name

Transfer Method: Byte Array
Stdout

```
Capture-HPC Wrappper. Version 2.0
[2009/03/26 15:17:53] INITIATED
[2009/03/26 15:17:53] PROCESSING
[2009/03/26 15:19:14] PROCESSING - 55%, visited=5, safe=5,
malicious=0, error=0
```

Figure 4.9: Screenshot of our Web portal accessing the ClientHoneypotWrapper service, encapsulating Capture-HPC.

these methods to the workflow. The previous version, Taverna 1.7.1, does not have this issue.

- The lack of WSRF support in 1.7.1, however, meant we had to change all the service context methods to pass the resource key as a parameter rather than the SOAP header as WSRF services should. When the resource key is explicitly passed the functionality remains the same, meaning it is more of an inconvenience that we lose the transparency of WS addressing.

- Another bug is that Taverna creates a malformed SOAP request for any method that doesn't have any parameters, throwing a security exception. To work around this we had to again change the WSRF services to add dummy parameters to any methods with empty parameter lists.

- We found that once we started processing a list of URLs using the *runClientHoneypotSh* method we could not get regular status updates on the progress. This was partially due to the lack of explicit iteration in Taverna, but even with a workflow language with explicit iteration—such as BPEL—this functionality would be difficult to achieve. This is because a workflow is like a service and only outputs values at completion, so intermediate values would need to be passed to another service. So this type of publish/subscribe mechanism for notifications from services is a feature lacking from all workflow systems, not just Taverna. To get around this issue we had to write the resource reference for the service context to a file using a BeanShell script, the user is then able to input it into the Web portal to check progress.

- A limitation of the BeanShell implementation in Taverna 1.7.1 is the object types of the inputs and outputs into the BeanShell processor. These inputs and outputs must be specified from a predefined list and are variations of strings and binary data, multiplicity can be specified as multi-dimensional lists[1]. This limits the usefulness of the BeanShell system as a number of Web services will use custom types and other primitive types such as arrays. We have found that other types can be output from these beanshells (ie. an array can be passed out even if the output was specified as a list) because it just passes a reference to the object and the next processor can cast it as needed. This does not work with the inputs as they are declared as their Taverna types before they are assigned values, so would cause class cast exceptions if cast to any other types. Taverna 2.0 addresses this issue.

- We found the process of workflow creation quite tedious. For each service method used, XML splitters for input had to be added and uniquely named. The local services (built-in functions) lacked simple operations, eg. converting a list (which Taverna uses for all collections) to an array, meaning these operations had to be written as new BeanShell scripts. These scripts had to be written multiple times—if needed more than once—as Taverna does not allow you to reuse them.

Figure 4.10 shows the Taverna implementation of our earlier designed workflow—for basic operation of a client honeypot—that we gave in Figure 4.5. The service calls (in green) are matched, the additions are the XML splitter (in purple) and BeanShells (in brown) for splitting and parsing the data. Specifically, BeanShell scripts were needed for parsing filepaths to filename strings, loading the files into Base64 encoded byte arrays, converting ArrayLists to arrays, and writing the resource reference to file. For simplicity this diagram omits some details, such as the individual ports on the XML-splitters and string constants.

Figure 4.11 shows the Taverna implementation of our earlier more complex workflow for processing URLs on two differently configured systems—originally shown in Figure 4.6. In this workflow we utilise some of the sections from the previous Taverna workflow—method calls, XML-splitters, and

[1]In Taverna they are called lists though they are actually the Java ArrayList type.

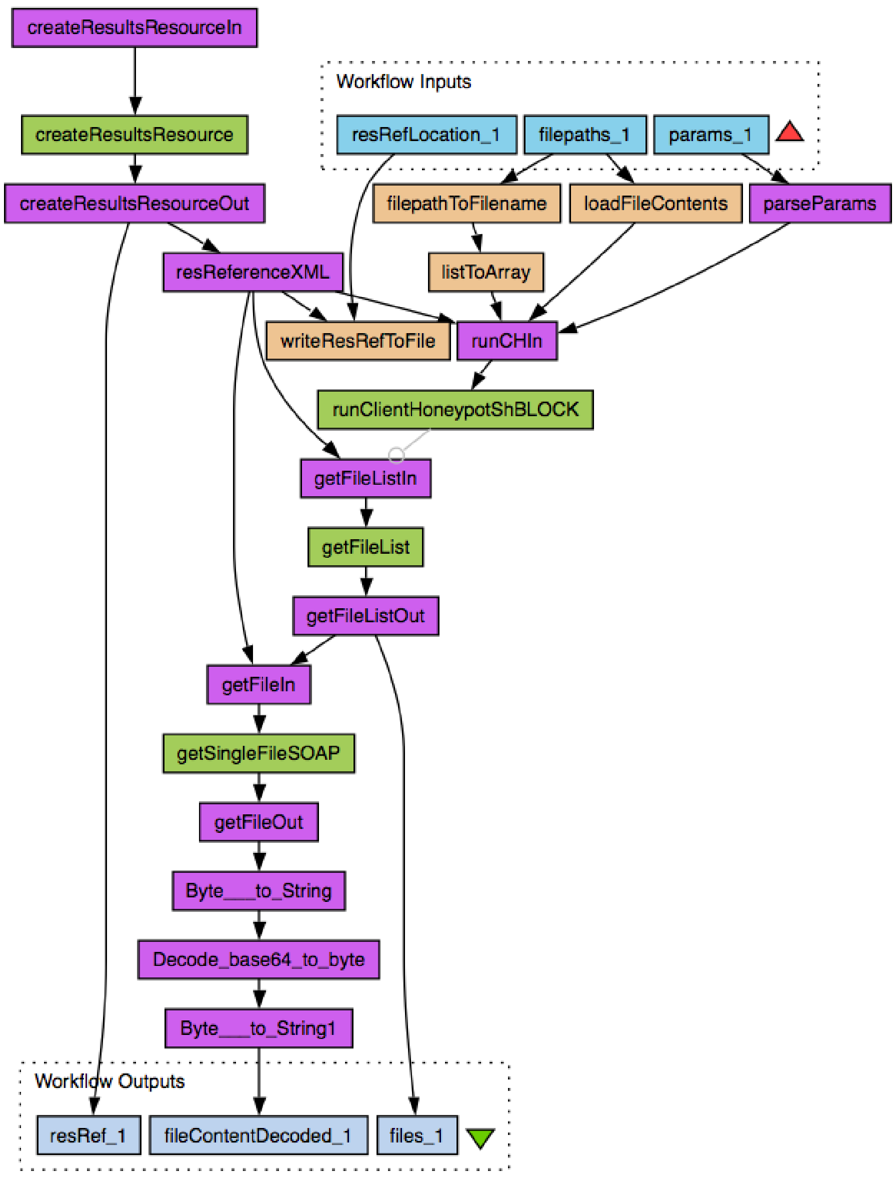


Figure 4.10: Taverna workflow modelling the basic operation of a client honeypot.

BeanShells—through the use of nested workflows.

4.5 Summary

In this chapter we have taken the requirements we formulated in Chapter 3 and designed a user-oriented, loosely-coupled and Grid-integrated system using WSRF services and workflows.

We investigated the areas that workflows can be applied to client honeypots, finding that they are useful for both operating a specific client honeypot, and the federation of multiple client honeypots. Our analysis showed that these two areas have different requirements for workflows languages, the first being more suited to control-oriented languages such as BPEL, and the second suiting data-oriented languages such as Taverna. The expressibility of both languages was assessed by graphically modelling sample workflows.

From our overall design we implemented a prototype system, encapsulating Capture-HPC—our case study client honeypot—with a shell script and Grid services created with Introduce and gRAVI. Access to the Capture-HPC functionality is provided through a Web portal. Taverna workflows were created to implement the sample use cases we described in Section 3.6.

In the next chapter we will evaluate our system, determining if it meets the requirements and whether it is an improvement on alternative system designs.

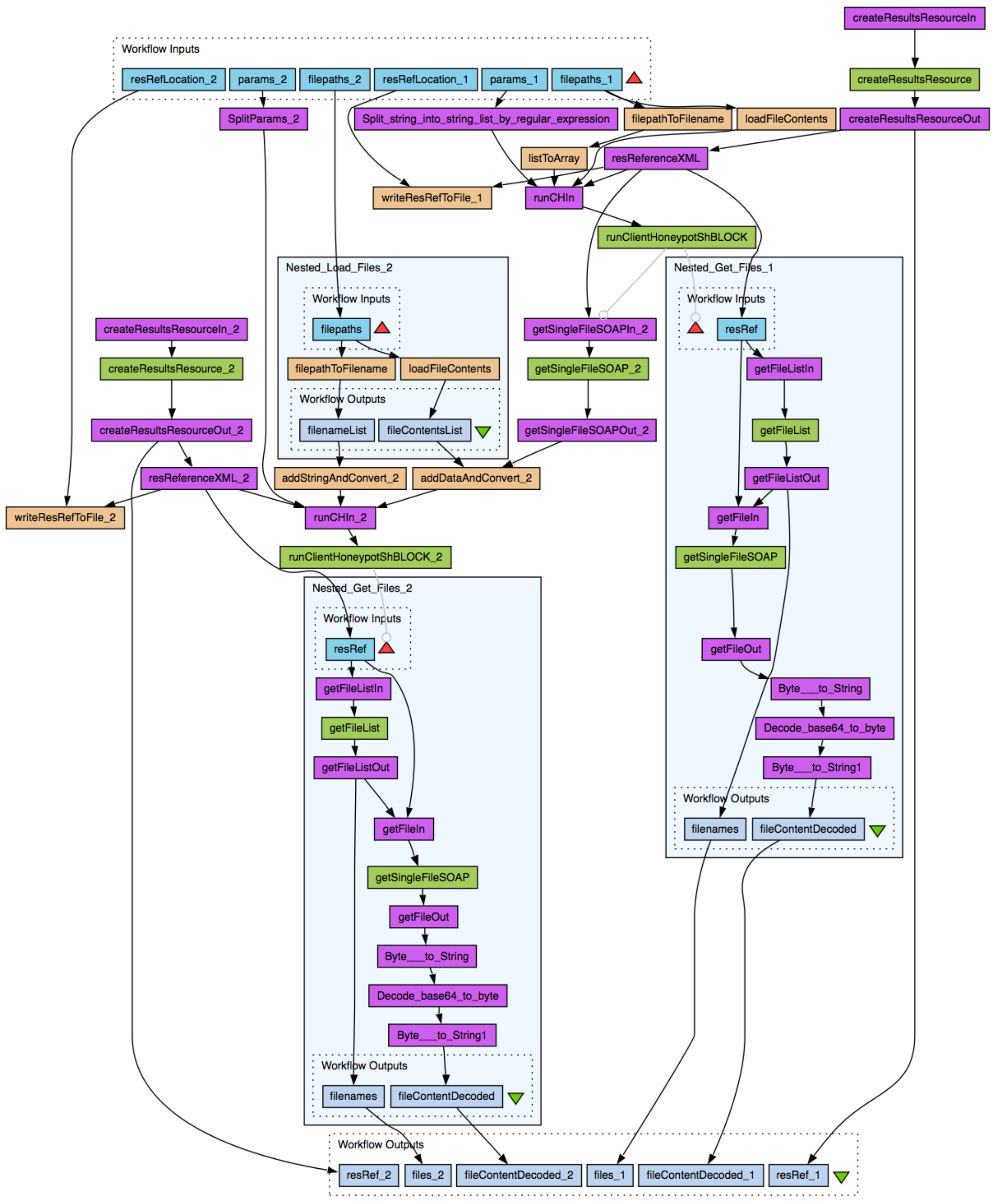


Figure 4.11: Taverna workflow implemented for processing URLs on two differently configured systems.

Chapter 5

Evaluation

This evaluation chapter covers a number of different aspects of our system.

Firstly, in Section 5.1 we assess how well our original system requirements are met.

One of the most important aspects of our system is the workflow component, used for modelling and managing tasks. So in Section 5.2 we define a set of criteria to enable us to make qualitative assessments of workflow systems, including Taverna and alternative designs. These alternatives include a manual script-based system and a Java system that uses our WSRF services. We also evaluate the authoring tools used to create these services.

Section 5.3 evaluates our Taverna workflow against the two alternatives. As part of this evaluation we created a Java application, operating on our WSRF wrapper services—implementing one of our Taverna workflows.

In Section 5.4 we look at the wider issue of applicability of workflow systems to our domain, including missing features and requirements from other Internet instruments.

Finally, in Section 5.5 we summarise our findings.

5.1 Fulfilment of Requirements

In this section we evaluate our system to determine how well it meets the requirements we formulated in Chapter 3. These were derived from our experience of manually operating Capture-HPC for moderate scale scans over a period of seven months. In Section 4.4 we showed how our implementation satisfied the use cases we had created in Section 3.6. We focused our research on the key requirements—a system that is user-oriented, loosely-coupled and Grid-integrated. We now assess how well we have met each of these requirements.

We aimed to design a user-oriented system that focused on how the user wants to interact with the system, hiding unnecessary detail. Our Taverna workflows have addressed the use cases we specified in Section 3.6. Through our Web portal users can access client honeypots much more easily—previously users had to log onto the machine physically or remotely via SSH. All configuration options are optional and are standardised between client honeypots where possible, thus abstracting specific system detail. Choice is provided to users of variable levels of detail in the status updates given via the Web portal. Our ClientHoneypotManager component provides a user-oriented distributed broker with indexing services so that users can simply submit a URL list and system specification to the broker service which will deal with the actual allocation.

Secondly, we were looking for a loosely-coupled system, providing interoperability of client honeypot systems. Our WSRF Web services provide standard interfaces to wrap any client honeypot system, including both high and low-interaction versions. No changes need be made to existing client honeypots if either an intermediary custom shell script is written or an API is exposed.

Thirdly, we wanted a system that is Grid-integrated, to utilise the many Grid resources available, provide a flexible de-centralised architecture and promote interoperability over multiple sites. Our system uses WSRF services, allows GridFTP file transfers, and has a de-centralised design. It is able to be integrated into GRAM for resource managers to control allocation of VM resources. Underlying Grid security can be implemented via options in the service container, allowing the services to be potentially securely used within a virtual organisation.

5.2 Evaluation Methodology

Our methodology was to develop evaluation criteria (shown in Section 5.2.1), implement three versions of workflows—using shell scripts, Taverna and Java—and evaluate this experience. The Taverna and Java versions both have shared features—the WSRF services created with the authoring tools Introduce and gRAVI—which we evaluate in Section 5.2.2.

5.2.1 Criteria

We have devised the below criteria covering six important areas for evaluating our use of workflow technologies, the aim being to use this to perform qualitative assessments of each workflow version (shell scripts, Taverna and Java), allowing comparisons to be drawn.

- *Required knowledge*—the technical knowledge of the tools a user of the system requires to create and modify a client honeypot process, for instance in our system the required knowledge will relate to Taverna and workflows languages in general. This also includes any low-level knowledge of specific client honeypots if this is not abstracted.
- *Modifying workflows*—this will be an analysis of the effort required to modify existing client honeypot workflows. A possible change could be an existing process that feeds data from one client honeypot to another and we want to modify this to anonymise the data. Our assessment would be the effort required for this change.

- *Maintainability*—the ease with which users can maintain these workflows. This is particularly important for complex workflows, because errors become more likely as complexity increases. Mechanisms such as dividing into components will help the maintainability.

- *Extensibility*—the ability to extend the core functionality of the workflow. Unlike modifying workflows, this is adding significant functionality to the end of the workflow. Of particular interest is the means to extend the workflow to include data analysis, eg. adding a function that generates some meta data to each malicious URL, such as the IP address of the Web server.

- *Understandability*—the ease with which a user can interpret a new process introduced to them, particularly important for complex processes. A graphical representation will help understandability.

- *Dealing with failure*—this is the ability to handle errors and recover to carry on operation if possible. Of interest are errors that occur related to client honeypot operation, eg. if an invalid configuration option is given, or a VMWare Server crashes. Ideally there will be fine-grained control of errors.

5.2.2 Shared Features

In this section we evaluate the tools we used to generate our WSRF Grid services—Introduce and its plugin gRAVI.

Prior to using Introduce we had looked into the development of Web services in Java. Even with a number of APIs available for Web services it did not appear a straightforward solution, particularly as we wanted to use WSRF services which are not well supported. The Introduce toolkit generated thousands of lines of code, saving us a significant amount of time. We found Introduce to be a stable Java application with a well-constructed GUI for defining and deploying the services.

By using the gRAVI plugin we saved further time, as this tool generated hundreds of lines of code. The code for handling file transfer, both with SOAP and GridFTP was particularly useful. From taking a number of days to write basic service skeletons, we could create full services in a number of hours. This follows similar experiences of another researcher who had spent over a month developing basic Web services for accessing a physics application, and later used gRAVI to create full services in less than two days [43].

A feature that would improve Introduce would be an integrated data type creation tool that automatically updates the types used the service. Currently, custom data types have to be created manually using XML-schema documents, and if changes are made to existing data types the affected methods have to reconstructed.

A few minor features which would improve gRAVI are: SOAP attachments, and delegation as part of the GSI security.

5.3 Evaluation against Taverna Workflows

We now compare our Taverna workflow system to alternative system designs—a manual system using shell scripts and a Java-based system that uses our WSRF services—using the criteria we presented earlier.

In Section 5.3.3 we summarise this semi-taxonomy with a table that compares the three alternative systems, assigning a star rating to each criterion.

5.3.1 Manual System with Scripts

In this section we compare our system's Taverna workflow design and enactment to the previous manual and script-based approach—previously used for the management of client honeypots. The following is our comparison using each of the criteria we stated in Section 5.2.1:

Required knowledge: The manual approach requires shell scripting experience and detailed knowledge of the specific client honeypot being used. Our workflow approach requires knowledge of workflows and Taverna but no knowledge of specific client honeypot details, such as configuration options, components, and data formats.

So, in terms of scalability, when using our workflow system the required knowledge would remain static if we increased the size of the modelled process to include the operation of two or three different client honeypots. However, if using a script-based system, the technical knowledge would increase, with details of each client honeypot system needed to be known.

Modifying workflows: There is a similar amount of effort required for both approaches. Changes would require lines of script changed or processors changed in a workflow. These changes could mean either the script parameters and return values could change, affecting other scripts; or workflow inputs and data flow changes could affect sub-workflows and super-workflows.

For example, adding an anonymising function would require a new shell script to be written with the input data types matching the first client honeypot's output, and the script's output data types to match the second client honeypot. The top level control script would need to be modified to call this new script and deal with passing the data. The workflow approach would require a new anonymising processor between the the two client honeypots, either as a BeanShell script or a service call, and the appropriate data flow lines changed.

Maintainability: Any shell scripts will be specific to one particular client honeypot system and usually a specific deployment. Therefore, there is a lack or code sharing and testing which adds to the maintenance overhead. Scripts can be more error prone to bugs in the code (eg. a missed parameter passed) and require documentation.

Workflows can be shared and applied to other systems with little, or no, modification. A data flow language like Taverna is easier to match inputs and outputs of processors, though the type checking is limited and BeanShell scripting is error-prone.

Extensibility: Scripts are well-suited for pipelining simple text but not complex multi-value data types which have to be precisely matched from one output script to the next input script. Larger scale data analysis is suited to parallel operations and Grid processing, supported by workflows but not simple shell scripts.

Understandability: Small-scale scripts are easily understood, but for more complex processes the scripts will need to be divided into components and well documented to remain understandable. The graphical representation of our workflows communicates the process effectively (until we get to large processes), particularly the data flow between processors and sub-workflow components.

Dealing with failure: Neither scripts or Taverna workflows deal with failure particularly well. Both will output what error has occurred but it is difficult to code any error handling into the processes. Alternative workflow languages, such as BPEL, offer better exception handling than Taverna.

In summary, for small-scale processes—operating on a client honeypot that the user is familiar with—the scripting method is adequate. But for any larger scale processes using more than one client honeypot, requiring code reuse or data processing, our workflow approach is significantly better suited.

5.3.2 Java & Web Services System

In the previous section we showed how our system design was an improvement on the existing manual management method of using shell scripts. We now evaluate our system against another alternative system design.

The service-oriented architecture of our design is effective for a loosely-coupled and Grid integrated system. Therefore, the WSRF services for wrapping and managing client honeypot systems appear a solid design with few obvious alternatives. However, the effectiveness of the mechanism we use for managing the processes that consist of these services (the workflow system in our design) is less clear and requires further evaluation.

An alternative system design is a Java application that controls the orchestration of Web services. The Java application can then use our WSRF wrapper services to control client honeypots, combining these services to form complex processes in the same way workflows do.

As part of this evaluation we wrote a basic Java application that implements our workflow in Figure 4.11 for processing URLs on two differently configured systems[1]. This was relatively straightforward, with less than a hundred lines of code needed—this was partly because we could use some of the Introduce generated client code to hide service invocation detail. The creation of this Java application provides a point of comparison to our Taverna workflow designs. High-level scripting languages such as Ruby and Python would be alternatives to our use of Java.

Rather than a standalone application, this type of system could also be a JSP (Java Server Page) based Web application, or deployed as a Web service the same way workflows can be exposed in languages such as BPEL.

We now compare this Java application design to our Taverna workflow approach, again we use the criteria we defined in Section 5.2.1:

Required knowledge: A Java application approach to coding the service would require only reasonable knowledge of Java programming and some knowledge of Web service invocation. The workflow approach requires quite detalied knowledge of the workflow language and the tool used to design the workflows. Data flow languages such as Taverna are simpler to learn than BPEL (due to the lower complexity and integrated

[1]The code for this application is provided in Appendix D

graphical editors), but are still not straightforward, and our experience has shown that Java code is often required inside the workflow via BeanShell scripts.

The Taverna workflow design tool is a stand-alone application. This means it will be separate to the Web portal interface we use for system access and no customisations can be made to the workflow design system to make it more domain specific. Whereas, a Java application could have its own custom interface for modifying the processes, and target this towards client honeypots. If this type of dynamic application was used it would also remove the need for any Java knowledge to use such a process-modification system, though this would require significant development work.

Modifying workflows: A well-designed Java application should generally not require great effort to modify an existing process, both for small and large changes. This is because objects, methods and variables can easily be added, removed, or modified, and the strongly-typed system will ensure the changes are well-formed.

Small changes to a Taverna workflow, such as adding a new processor between two existing processors, can be achieved quite easily as this just requires the modification of input and output data-flows. Larger changes are more difficult because the data flow will need to be preserved but may need multiple changes which is error-prone without a strongly typed system, and the GUI based system limits changes such as “cutting” and “pasting” sections of a workflow.

For example, changing a parameter in a Web service method’s signature from a `boolean` to an `int` would require only a text search and replace to change the Java method calls and the generated stubs to be rebuilt. However, using the Taverna GUI would require the WSDL to be re-added, and each of those existing method processors and XML-splitters deleted and the new version re-added and associated XML-splitters created. The underlying XML of the workflow could be edited by hand outside Taverna, though it would not be desirable to expect users to have to do this, and be error prone for large modifications.

Maintainability: Both system designs should be quite simple to maintain because they can both be divided into manageable components. The Java application approach is better suited to debugging, with integrated debuggers available. Debugging in Taverna is limited, break-points can be set but we cannot inspect data-flow values or step through the BeanShell scripts.

Error checking is also an advantage for compiled Java application code, compared to the interpreted code in Taverna.

Extensibility: The componentisation of both system designs will allow extensibility to be easily implemented. However, Taverna is better suited for scientific data analysis because of the mechanisms available for pipelining, parallel execution and Grid integration. Data analysis is an important part of the client honeypot process, and data gathered with high-interaction client honeypots can be used to formulate the signatures used in the high-speed low-interaction client honeypots [34].

Understandability: It is questionable which approach is more understandable and probably depends on individual users' preferences—some may prefer a pure code view, while others may prefer a graphical model. An advantage of the pure code view of a Java application is the ability to easily add comments to code to help readability, which is not possible in Taverna. Another consideration of the graphical approach is that there is still code in the processes, located in BeanShell scripts, which takes more effort to access than a complete code listing. Some of the drawbacks of the graphical approach are due to the immaturity of the tools and improvements are likely to be made.

Dealing with failure: Java has a much better exception handling system which gives fine-grained control of recovering from errors arising from client honeypots. Taverna has poor error handling and workflows will exit upon any errors encountered with no recovery options available. For example, if a data analysis service went offline, in Java the exception handler could skip this service (or use an alternative), but in Taverna the entire workflow would exit after the specified number of service retries failed.

Taverna relies on error handling to be implemented in the services themselves. Tools external to the workflow may need to be used to recover from error and start processing again. For example, if an exception was thrown before the workflow could retrieve and parse the output files from a client honeypot, then this may have to be done by another clean-up workflow, or manually.

To summarise these findings, we have determined that the Java application approach improves on required knowledge, modifying workflows, dealing with failure, and slightly better for maintainability. The undertandability of both approaches is quite even, though improvements to Taverna's GUI could change this. The area that our Taverna workflows have an advantage is extensibility, specifically the data analysis phase.

5.3.3 Comparison Summary

Table 5.1 summarises our findings, comparing the three system designs on the five identified criteria. We assign a star rating to show if that criterion is a strength or a weakness.

One area that we have not yet covered is performance in terms of processing speed. Because high-interaction client honeypots need to wait a number of seconds per URL to determine state changes—ten in our experiment—any additional processing will be a small fraction of this delay, so we do not see computational overhead/performance as a critical factor. The script-based approach differs because the shell scripts are generally run locally, whereas

Table 5.1: Comparison of system designs. Each criterion is given a star rating—one star is weak, three is satisfactory, and five is strong.

	Manual script-based	Taverna	Java application
Required knowledge	★★	★★★	★★★★
Modifying workflows	★★★	★★★	★★★★★
Maintainability	★★	★★★	★★★★
Extensibility	★★	★★★★★	★★★
Understandability	★★	★★★	★★★
Dealing with failure	★	★	★★★★★

the other two Web service approaches have the overhead of protocols (such as SOAP) that will be slower. However indirect performance benefits are gained through distribution, the ability to offload to remote providers, simple parallelism through multiple services, and Grid computing.

5.4 Generalisation of Evaluation

In the previous sections we have evaluated our system against manual script-based operation and a Java application approach, particularly looking at our Taverna workflow approach. We now want to take a step back and look at how applicable workflow systems in general are to our domain, client honeypots specifically, but also more widely for Internet instruments.

Workflow technology appeared promising because it is a system that is dedicated to controlling processes constructed of Web services, with the control-oriented language BPEL widely used in the business sector, and a number of data-oriented languages used in scientific areas. These workflow systems are presented as options for non-technical users to be able to manipulate complex service-based processes, but our experience suggests that even relatively simple workflows need significant low-level code to work. An illustration of this is the differences from our preliminary workflow designs we created to their actual implementations. Figure 4.6 showed our preliminary workflow for modelling processing URLs on two differently configured systems, Figure 4.8 was the actual Taverna implementation. The first model shows 15 processors, but the implementation has 46 processors. This is due to additions for the XML-splitters, local processors for doing basic type conversions, and nine BeanShell scripts for doing other conversions and loading files. This changes a relatively straight-forward model into quite a complex one, with increased effort both to construct and to understand it.

In Section 4.4.3 we outlined a number of issues we encountered using Taverna. This is somewhat understandable given that workflows are a relatively new area of research, with a large number of systems and little convergence, especially in the data-oriented languages. We will now identify some useful functionality not already in workflow systems.

The main piece of functionality we would like to see is a publish/subscribe mechanism for workflows to get status updates from WSRF services and notify users. Services generated by Introduce provide subscription options so that a service user can provide an endpoint reference so they can receive call-backs on resource state changes. A mechanism could be provided in the workflow language to subscribe to this type of method and specify how any notifications should be dealt with, ie. notifying the workflow user. This type of functionality would allow us to subscribe a workflow to a client honeypot while is has invoked a processing job, providing updated status and possibly perform actions depending on upon this status.

More minor functionality we would like to see includes better exception handling, automatic code generation (such as for XML-splitters), an integrated debugger, easier conversion of types, and an improved GUI with easily adjustable levels of detail for displaying a model.

Some of our experience is driven by the type of application we are developing for, and the workflow system we have used. Therefore, we acknowledge that generalisation of our experiences may be limited. We have not found any other published experiences of workflow usage in the same or similar domains, so there is little to compare our findings with.

The domain we are interested in appears to have elements of both control-oriented and data-oriented workflow system, but these are currently divergent focuses of workflow systems. If this is not a unique requirement, a combined approach to workflow systems could be developed. The maturity and design of BPEL would appear to limit changes in orientation, so it may be more feasibly for scientific workflow languages such as Taverna to become more control oriented—a first step would be explicit iteration.

It is interesting to look at other Internet instruments to determine what requirements they would have from workflows. A network telescope would appear to fit a data-oriented workflow language due to its passive data collection requiring very little control. This contrasts with the active data collection requiring a high level control for a client honeypot system. A network telescope requires significant data processing to get useful information so this suits the parallel and pipelined nature of a data driven workflow.

5.5 Summary

In this chapter we have conducted a number of evaluations, we now briefly summarise each.

We evaluated the Grid service authoring tools we used, Introduce and gRAVI, finding them solid toolkits for service creation, saving developers significant time and effort.

Next, we compared our Web service and Taverna workflow system to our experiences to the manually managed script-based system we formed our requirements from. We found that our system satisfied the identified requirements of being user-oriented, loosely-coupled and Grid-integrated. For anything apart from small-scale workflows our system scored better for the criteria we assessed upon than the script-based operation.

We created a Java application that enacted our WSRF services as an alternative system design for comparison. We found that the Java approach was better suited on most of the criteria we assessed upon—modifying workflows, maintainability, and dealing with failure. The main area that Taverna workflows showed an improvement was extensibility with good support for data analysis through pipelining. Also, Taverna was good in terms of usability—with a drag and drop GUI, minimum knowledge needed of underlying technology, and the ability to pass a workflow to a colleague which will work in their Taverna.

The choice of which approach should be taken—Taverna or Java—will depend upon the requirements of a specific project, ie. weighing how important integrated data analysis mechanisms are compared to fault-handling and the required technical knowledge for users.

Finally, we used our experiences with workflows to assess their applicability to our domain. We found that, with some improvements, workflows could be beneficial to our research area, particularly with the end-to-end control of processes, from data collection to data analysis. Workflow technology may however be better suited to Internet instruments with passive data collection, such as network telescopes. We have identified some functionality that would improve workflow systems, most significantly a publish/subscribe system for workflows to monitor the state of WSRF resources.

Chapter 6

Conclusion

6.1 Summary of Thesis

The goal of this research is to enhance the previous manual script-based approach to client honeypot management. Our prototype uses WSRF services for wrapping client honeypots, Taverna to model and enact complex workflows, and a Web portal for access.

Our evaluation showed that while our system design satisfied our functional requirements, a Java-based application operating on our WSRF services provides a number of advantages over the Taverna approach used in our system—particularly for modifying workflows, maintainability, and dealing with failure. However, Taverna workflows are better suited to the data analysis phase of client honeypot operation and have some usability benefits—such as a drag and drop GUI, minimum knowledge of underlying technology, and the ability to easily share workflows. Workflow languages such as Taverna are still relatively immature, so improvements are likely to be made. Both of these approaches are significantly easier to manage and deploy than the previous script-based method.

We now attempt to answer our two research questions. The first was:

How can we effectively create systems of different client honeypots

> *to perform measurements and analysis?*

Our research indicates that using WSRF services with standard interfaces designed to wrap a generic client honeypot allows simple integration of different existing systems, with minimal development effort. The standard interfaces allows a manager component to act as a broker to users, abstracting unnecessary system detail.

The second question was:

> *What is the best method to automate the various tasks that encompass client honeypot management?*

Both Taverna and a Java application—operating on our WSRF wrapper services—provide good methods of client honeypot management. The Java approach could be considered better, but this judgment is largely dependent on factors of specific projects, such as the level and structure of data analysis and the benefit of a graphical notation.

6.2 Overall Contributions

The main contributions of this research are:

1. Creation of use cases to model system requirements of a client honeypot automation system, based on our experience of manually managing a system for moderate-scale scans over a period of several months.

2. Creation of WSRF Web services for wrapping any client honeypot, with minimum effort. These can be used by workflow languages and Java-based applications.

3. Implementation of workflows that use our services to model and control complex processes used in client honeypot operation.

4. Experience with Introduce and gRAVI, two recently developed Grid service authoring tools.

5. Evaluation of the Taverna workflow system for modelling and enacting client honeypot workflows–finding that Java was better for modifying workflows, maintainability, and dealing with failure; while Taverna had advantages in extensibility and areas of usability.

6.3 Future Work

The research we have conducted leads directly and indirectly to a number of pieces of future work, this includes:

- Implement the ClientHoneypotManger component in our system.
- Expand the current workflows to include data processing and analysis functionality after data collection.
- Expand our system implementation to operate over multiple organisations, with required security mechanisms such as delegation of credentials.
- Modify a workflow engine to support a publish/subscribe mechanism to Web service notifications, and output this data to users.
- Investigate the idea of a hybrid workflow system which has both control-oriented and data-oriented aspects.
- Use our architecture for other Internet instruments, ie. network telescopes.
- Investigate the use of cloud computing abilities into our architecture, eg. recruiting resources on demand.

Appendix A

Sample Capture-HPC Log

This appendix is a sample Capture-HPC log for a malicious URL. The site—`http://blackmores.co.nz`—was visited in September 2008. Each entry in the log contains: the type of system activity (file, process, or registry), the timestamp, and the data values relating to that system activity.

Of particular interest is: (1) the entry where the malicious executable is downloaded:

```
"file","25/9/2008 6:49:50.515","C:\Program Files\Inte
rnet Explorer\IEXPLORE.EXE","Write","C:\msntstza.exe"
```

(2) the entry for the executable being loaded as a new process:

```
"process","25/9/2008 6:49:50.561","C:\Program Files\I
nternet Explorer\IEXPLORE.EXE","created","C:\msntstza
.exe"
```

The full listing follows (some repeated entries have been removed for brevity):

```
"process","25/9/2008 6:49:49.983","C:\Program Files\Internet Explorer\IEXPLORE.EXE",
   "created","C:\WINDOWS\Temp\vv0QZXFd.com"
"file","25/9/2008 6:49:50.124","C:\WINDOWS\Temp\vv0QZXFd.com","Write","C:\WINDOWS\Te
   mp\twe1.tmp"
```

```
"file","25/9/2008 6:49:50.140","C:\Program Files\Internet Explorer\IEXPLORE.EXE","Wr
   ite","C:\WINDOWS\WindowsUpdate.log"
"registry","25/9/2008 6:49:50.202","C:\WINDOWS\Temp\vv0QZXFd.com","SetValueKey","HKL
   M\SOFTWARE\Microsoft\MSSMGR\Data"
"file","25/9/2008 6:49:50.233","C:\WINDOWS\Temp\vv0QZXFd.com","Write","C:\WINDOWS\sy
   stem32\winwpa32.dll"
"registry","25/9/2008 6:49:50.218","C:\WINDOWS\Temp\vv0QZXFd.com","SetValueKey","HKL
   M\SOFTWARE\Microsoft\MSSMGR\LSTV"
"registry","25/9/2008 6:49:50.218","C:\WINDOWS\Temp\vv0QZXFd.com","SetValueKey","HKL
   M\SOFTWARE\Microsoft\MSSMGR\Brnd"
"registry","25/9/2008 6:49:50.233","C:\WINDOWS\Temp\vv0QZXFd.com","SetValueKey","HKL
   M\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Notify\winwpa32\Asynchron
   ous"
"registry","25/9/2008 6:49:50.233","C:\WINDOWS\Temp\vv0QZXFd.com","SetValueKey","HKL
   M\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Notify\winwpa32\DllName"
"registry","25/9/2008 6:49:50.233","C:\WINDOWS\Temp\vv0QZXFd.com","SetValueKey","HKL
   M\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Notify\winwpa32\Impersona
   te"
"registry","25/9/2008 6:49:50.233","C:\WINDOWS\Temp\vv0QZXFd.com","SetValueKey","HKL
   M\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Notify\winwpa32\Startup"
"registry","25/9/2008 6:49:50.233","C:\WINDOWS\Temp\vv0QZXFd.com","SetValueKey","HKL
   M\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Notify\winwpa32\Shutdown"
"registry","25/9/2008 6:49:50.233","C:\WINDOWS\Temp\vv0QZXFd.com","SetValueKey","HKL
   M\SOFTWARE\Microsoft\MSSMGR\MSLIST"
"file","25/9/2008 6:49:50.515","C:\Program Files\Internet Explorer\IEXPLORE.EXE","Wr
   ite","C:\msntstza.exe"
"process","25/9/2008 6:49:50.561","C:\Program Files\Internet Explorer\IEXPLORE.EXE",
   "created","C:\msntstza.exe"
"file","25/9/2008 6:49:50.515","C:\Program Files\Internet Explorer\IEXPLORE.EXE","Wr
   ite","C:\msntstza.exe"
"process","25/9/2008 6:49:50.608","C:\Program Files\Internet Explorer\IEXPLORE.EXE",
   "terminated","C:\msntstza.exe"
"file","25/9/2008 6:49:50.515","C:\Program Files\Internet Explorer\IEXPLORE.EXE","Wr
   ite","C:\msntstza.exe"
"file","25/9/2008 6:49:51.640","C:\WINDOWS\Temp\vv0QZXFd.com","Write","C:\WINDOWS\Te
   mp\twe1.bat"
"process","25/9/2008 6:49:52.218","C:\WINDOWS\Temp\vv0QZXFd.com","created","C:\WINDO
   WS\system32\cmd.exe"
"file","25/9/2008 6:49:52.202","System","Write","C:\WINDOWS\Temp\twe1.bat"
"file","25/9/2008 6:49:52.218","C:\WINDOWS\Temp\vv0QZXFd.com","Write","C:\WINDOWS\Te
   mp\vv0QZXFd.bat"
"file","25/9/2008 6:49:52.218","System","Write","C:\WINDOWS\Temp\vv0QZXFd.bat"
"process","25/9/2008 6:49:52.233","C:\WINDOWS\Temp\vv0QZXFd.com","created","C:\WINDO
   WS\system32\cmd.exe"
"file","25/9/2008 6:49:53.811","C:\WINDOWS\system32\cmd.exe","Delete","C:\WINDOWS\Te
   mp\vv0QZXFd.com"
"file","25/9/2008 6:49:53.936","C:\WINDOWS\system32\cmd.exe","Write","C:\Program Fil
```

```
   es\Capture\logs\deleted_files\C\WINDOWS\Temp\vv0QZXFd.bat"
"file","25/9/2008 6:49:53.952","C:\WINDOWS\system32\cmd.exe","Delete","C:\WINDOWS\Te
   mp\vv0QZXFd.bat"
"process","25/9/2008 6:49:53.999","C:\WINDOWS\Temp\vv0QZXFd.com","terminated","C:\WI
   NDOWS\system32\cmd.exe"
"file","25/9/2008 6:49:53.983","C:\WINDOWS\system32\cmd.exe","Delete","C:\WINDOWS\Te
   mp\twe1.tmp"
"process","25/9/2008 6:49:54.77","C:\WINDOWS\Temp\vv0QZXFd.com","terminated","C:\WIN
   DOWS\system32\cmd.exe"
"file","25/9/2008 6:49:54.61","C:\WINDOWS\system32\cmd.exe","Write","C:\Program File
   s\Capture\logs\deleted_files\C\WINDOWS\Temp\twe1.bat"
"file","25/9/2008 6:49:54.61","C:\WINDOWS\system32\cmd.exe","Delete","C:\WINDOWS\Tem
   p\twe1.bat"
```

Appendix B

Manual Operation Scripts

The following are brief descriptions of the shell scripts we created for the manual operation of Capture-HPC (as described in Chapter 3):

- **check_server_csv.sh**

  Every 60 seconds prints out stats on the current CaptureServer progress. The data includes number of URLs visited, number of malicious URLs, safe URLs, and error counts. The data is in CSV format so the data can easily be piped to a file and opened in a spreadsheet application.

- **clear_pdnsd_cache.sh**

  Clears the PDNSD cache. Must be run as root. The old cache is copied to *archived_caches* in the proxy users home directory.

- **clear_squid_cache.sh**

  Clears the Squid cache. This will stop and restart the Squid service. The old cache is copied to *archived_caches* in the proxy users home directory.

- **combine_logs.sh <directory-containing-partial-logs>**

  This script is used to combine the log files from a number of partial logs into a combined log for each type of log (ie. error.log, malicious.log, progress.log, and safe.log). The directory passed as a parameter should contain only these partial log directories.

The script will remove all the empty log files that CaptureServer creates for safe URLs (which could be hundreds of thousands) so can take some time to complete.

- **push_vm.sh <vm-image-directory>**

  Pushes the VM image directory (passed as a parameter) to all the clients specified in the script and installs these into VMWare Server, which involves registering the VM, powering it on and creating a snapshot. The script uses key-based `scp` so the user needs to add their identity to the authentication agent be issuing the `ssh-add` command prior to running this script.

- **start_and_keep_resuming_server.sh <input-urls-filepath>**

  This starts CaptureServer using the `start_server.sh` script below, passing it the URL file parameter. Then every 15min it checks that the server is still progressing. If its not (indicating CaptureServer has probably crashed), it will call the stop_server_move_log.sh script, which will kill the CaptureServer process and moves the logs into the CaptureServer directory, with the number of remaining URLs to visit appended to their name). If there no are more URLs left to inspect, the server will be restarted. NOTE: When the server restarts, the <input-urls-filepath> is truncated to only those URLs remaining to be visited. Therefore the script makes a copy of the input-urls file with ".original" appended.

- **start_server.sh <input-urls-filepath>**

  Starts `CaptureServer.jar` (with the specified URL file) and passes it some standard parameters, these include the memory usage (`-Xms512m -Xmx1024m`) IPStack (`-Djava.net.preferIPv4Stack=true`), and server IP address/port (`-s 10.0.0.1:7070`).

- **start_squid.sh**

  Starts the Squid proxy cache as a background process.

- **stop_server_move_log.sh <input-urls-filepath>**

  Generally only used as part of the start_and_keep_resuming_server.sh script. Kills the CaptureServer process and moves the logs into the CaptureServer directory, with the number of remaining URLs to visit appended to their name. It then truncates the input URL file to only the remaining URLs or indicates if there are no more URLs to inspect.

Appendix C

Service Interfaces and Data Types

This appendix specifies the WSRF interfaces for our two Grid services, and subsequently the complex data types used.

The ClientHoneypotManager interface is defined:

- CHSessionManagerReference initiateSession ();
- void registerClientHoneypot (ClientHoneypotDescriptor info);
- void deregisterClientHoneypot (ClientHoneypotDescriptor info);
- String processURLs (int priority, ClientHoneypotConfig settings, Base64Binary urlListFile, Reference endpointRef);
- void terminateProcessing ();
- SessionStatus getStatus ();

The ClientHoneypotWrapper interface is defined:

- ResourceReference createResultsResource ();
- ResourceReference runClientHoneypotSh (ResourceReference resRef, String[] arguments, String[] filenames, Base64Binary[] inputFiles);

- String[] getFileList();
- Base64Binary getSingleFileSOAP(String filename);
- Status getStatus();
- boolean killProcess();
- boolean sendFileSOAP(String filename, Base64Binary inputFile);
- String[] getAllFilesGridFTP();
- boolean getSingleFileGridFTP(String filename);
- String getDir();
- State getStateRP();

We now present the complex types used in the previous WSRF services in a readable form, they are formally defined in a XSD file. There are a number of enumerated types included, the values will be defined in any implementation and relate to a specific namespace. Within a complex type definition we use '[ ]' to define an optional type and '*' to show that more than one of the indicated types may be included. The complex types are:

- **ClientHoneypotDescriptor**: int clientHoneypotType, String clientHoneypotName, String location, int country, [int numberNodes], *[DataPair additionallData]
  1. *clientHoneypotType* is the classification of the client honeypot system, eg. `HIGH_INTERACTION`.
  2. *clientHoneypotName* is the system's name, eg. "Capture-HPC".
  3. *location* is the URI of the system.
  4. *domain* is the domain where the system is located, eg. ".nz".
  5. *numberNodes* is relevant if the system comprises of multiple nodes.
  6. *additionallData* is any other searchable information to be stored.

- **SystemDescriptor**: int OS, [int OSVersion], *(int browser), *[int browserPlugin], *[int clientApp]

  1. *OS* is the Operating System on the client system, eg. `WINDOWS_XP`.
  2. *OSVersion* is the specifc version of the OS, eg. `SP2`.
  3. *browser* is a Web browser installed on the client system, eg. `FIREFOX_3_1`.
  4. *browserPlugin* is any plugin installed in the Web browser, eg. `FLASH_9`.
  5. *clientApp* is any other applications installed in the client system, eg. `ACROBAT_9`.

- **ClientHoneypotConfig**: int clientHoneypotType, [SystemDescriptor clientSystemType], [int visitTime], *[DataPair additionalConfig], *[int pageMetaDataToCollect]

  1. *clientHoneypotType* is the classification of the client honeypot system, eg. `HIGH_INTERACTION`.
  2. *clientSystemType* is a descriptor of the client OS and applications.
  3. *visitTime* is an option for high-interaction client honeypots to specify the length of time to capture state changes after the Web page loads.
  4. *additionalConfig* is any system-specifc configuration options, suh as setting the Java heap size for CaptureServer.
  5. *pageMetaDataToCollect* is the name of any meta data attributes to collect for malicious pages, eg. `NUMBER_OF_REDIRECTS`.

- **Session**: String ClientHoneypotLocation, String resultsLocationURI, String creationTime, String sessionStatusReference

  1. *ClientHoneypotLocation* is the URI of the client honeypot system.
  2. *resultsLocationURI* is the location where the results files will be posted.

3. *creationTime* is the timestamp of the session.
4. *sessionStatusReference* is a reference to the status of the session.

– **SessionStatus**: int state, datetime startTime, int totalURLs, int visitedCount, int maliciousCount, int safeCount, [int networkErrors], [int processErrors], [int timeoutErrors], int totalErrors, *[DataPair additionalData]

1. *state* is an enumerated value indicating the current state of the client honeypot, eg. PROCESSING.
2. *startTime* is the timestamp of the current scan.
3. *totalURLs* is the total number of URLs being examined in the session.
4. *visitedCount*, *maliciousCount*, and *safeCount* are counts of pages processed.
5. *networkErrors*, *processErrors*, and *timeoutErrors* are counts of specific error conditions from the pages processed.
6. *totalErrors* is the total of all error conditions from the pages processed.
7. *additionalData* is any additional status information.

– **MaliciousList**: int count, *[URLData maliciousURL]

1. *count* is the number of malicious URLs in the list
2. *maliciousURL* provides data on an individual malicious URL. There should be a separate entry for every malicious URL counted.

– **URLData**: String url, *[DataPair metaData]

1. *url* is the URL that data is to be provided on.
2. *metaData* is meta data about the URL, such as the geographic location of the Web server.

– **DataPair**: String attribute, String value

1. *attribute* is the name of an attribute which has been measured.
2. *value* is the corresponding data value measured.

Appendix D

Java & Web Services System Source Code

The following is the sample code we wrote for the Java & Web services system (described in Section 5.3.2):

```
import java.io.File;
import java.rmi.RemoteException;
import org.apache.axis.types.URI;

//Introduce generated client code
import nz.ac.mcs.vuw.dsrg.geii.clienthoneypotwrapper.client.
   ClientHoneypotWrapperClient;
import nz.ac.mcs.vuw.dsrg.geii.clienthoneypotwrapper.context.client.
   ClientHoneypotWrapperResultResourceClient;
import nz.ac.mcs.vuw.dsrg.geii.clienthoneypotwrapper.context.stubs.types.
   ClientHoneypotWrapperResultResourceReference;

public class ClientHoneypotProcessor {

    public static void main(String[] args) {
        //Variables for testing
        String url =
           "http://10.73.2.133:8080/wsrf/services/cagrid/ClientHoneypotWrapper";
        ClientHoneypotWrapperResultResourceReference resRef1 = null , resRef2 = null;
        String pipedFile="safe.txt";
        String[] args1={"-u", "test-urls.txt"};
        String[] args2={"-u", pipedFile};
        String localResultsDir1="/Users/dave/Desktop/run-1";
        String localResultsDir2="/Users/dave/Desktop/run-2";
        String[] inputFilepaths1={"/Users/dave/Desktop/test-urls.txt"};
        String[] inputFilepaths2={localResultsDir1 + "/" + pipedFile};

        System.out.println("Starting processing 1");
        resRef1 = startProcessing(url, resRef1, args1, inputFilepaths1);
        getOutputFiles(localResultsDir1, resRef1);

        System.out.println("Starting processing 2");
        resRef2 = startProcessing(url, resRef2, args2, inputFilepaths2);
        getOutputFiles(localResultsDir2, resRef2);
    }

    public static ClientHoneypotWrapperResultResourceReference startProcessing(
       String url, ClientHoneypotWrapperResultResourceReference resRef, String[]
       arguments, String[] inputFilepaths){
        final ClientHoneypotWrapperClient client;

        try {
            client = new ClientHoneypotWrapperClient(url);
        } catch (URI.MalformedURIException e) {
            throw new IllegalArgumentException("Malformed URL: " + url, e);
        } catch (RemoteException e) {
            throw new IllegalArgumentException("Remote Exception", e);
        }

        //load files into Base64 arrays
        String[] fileNames = new String[inputFilepaths.length];
        byte[][] data = new byte[inputFilepaths.length][];
        File inputFile;
        for (int i=0; i<inputFilepaths.length; i++) {
            inputFile = new File(inputFilepaths[i]);
            fileNames[i] = inputFile.getName();
            data[i] = ClientHoneypotWrapperClient.encode(inputFile);
        }

        try {
           resRef = client.runClientHoneypotShBLOCK(resRef, arguments,
              fileNames, data);
        } catch (Exception e) {
            System.out.println(e);
            throw new RuntimeException("Error invoking command ", e);
        }
        System.out.println("resRef1:"+ resRef.toString());

        return resRef;
    }

    public static void getOutputFiles(String localDir,
       ClientHoneypotWrapperResultResourceReference resReference){
        ClientHoneypotWrapperResultResourceClient resClient;

        try {
            resClient = new ClientHoneypotWrapperResultResourceClient(
               resReference.getEndpointReference());
        } catch (RemoteException e) {
            throw new RuntimeException(
               "Initialization of Context Service client", e);
        } catch (URI.MalformedURIException e) {
            throw new IllegalArgumentException("Malformed URL: ", e);
        }

        try{
            String[] outFilenames = resClient.getFileList(resReference);
            for (int i=0; i<outFilenames.length; i++) {
                byte [] outputData = resClient.getSingleFileSOAP(resReference,
                   outFilenames[i]);
                if (outputData !=null) {
                    ClientHoneypotWrapperClient.decode(outputData, localDir,
                       outFilenames[i]);
                }
            }
        }catch (Exception e) {
            System.out.println("Error getting file  " + e);
            throw new RuntimeException("Requesting file: ", e);
        }
    }
}
```

Bibliography

[1] Akram, A., Meredith, D., and Allan, R. Evaluation of BPEL to Scientific Workflows. In *Cluster Computing and the Grid, 2006. CC-GRID 06. Sixth IEEE International Symposium on* (2006).

[2] Allcock, W., Bester, J., Bresnahan, J., Chervenak, A., Liming, L., and Tuecke, S. GridFTP: Protocol extensions to FTP for the Grid. *Global Grid ForumGFD-RP 20* (2003).

[3] Altintas, I., Berkley, C., Jaeger, E., Jones, M., Ludascher, B., and Mock, S. Kepler: An extensible system for design and execution of scientific workflows. In *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on* (2004).

[4] Amnuaykanjanasin, P., and Nupairoj, N. The BPEL Orchestrating Framework for Secured Grid Services. In *International Conference on Information Technology: Coding and Computing (ITCC'05)* (2005).

[5] Atkinson, I., du Boulay, D., Chee, C., Chiu, K., Coddington, P., Gerson, A., King, T., McMullen, D., Quilici, R., Turner, P., et al. Developing CIMA-Based Cyberinfrastructure for Remote Access to Scientific Instruments and Collaborative e-Research. *Australasian Symposium on Grid Computing and Research (AusGrid), Ballarat, Australia Conferences in Research and Practice in Information Technology, Vol. 6* (2007).

[6] Bagnasco, A., Poggi, A., and Scapolla, A. A Grid-Based Architecture for the Composition and the Execution of Remote Interactive Measurements. *e-Science and Grid Computing, 2006. e-Science '06. Second IEEE International Conference on* (Dec. 2006).

[7] Chard, K., Onyuksel, C., Tan, W., Sulakhe, D., Madduri, R., and Foster, I. Build Grid Enabled Scientific Workflows using gRAVI and Taverna. *SWBES08: Challenging Issues in Workflow Applications Workshop, Indianapolis, USA* (2008).

[8] Cherbakov, L., Galambos, G., Harishankar, R., Kalyana, S., and Rackham, G. Impact of service orientation at the business level. *IBM Syst. J., vol. 44, pp. 653-668* (2005).

[9] D. F. McMullen, T. D. Integrating Instruments and Sensors into the Grid with CIMA Web Services. *Proceedings of the Third APAC Conference on Advanced Computing, Grid Applications and e-Research (APAC05), Gold Coast, Australia* (2005).

[10] Dornemann, T., Friese, T., Herdt, S., Juhnke, E., and Freisleben, B. Grid Workflow Modelling Using Grid-Specific BPEL Extensions. In *German e-Science Conference* (2007).

[11] Erwin, D., and Snelling, D. UNICORE-a Grid computing environment. *Concurrency and Computation: Practice and Experience 14* (2002).

[12] Feller, M., Foster, I., and Martin, S. GT4 GRAM: A functionality and performance study. In *TeraGrid Conference* (2007).

[13] Foster, I. What is the Grid? a three point checklist. *GRID today 1*, 6 (2002).

[14] Foster, I. Globus toolkit version 4: Software for service-oriented systems. *Journal of Computer Science and Technology 21*, 4 (2006).

[15] FOSTER, I., KESSELMAN, C., NICK, J. M., AND TUECKE, S. The Physiology of the Grid. Tech. rep., Globus Project, Dec 2004.

[16] FOSTER, I., KESSELMAN, C., AND TUECKE, S. The anatomy of the grid: Enabling scalable virtual organizations. *Int. J. High Perform. Comput. Appl. 15*, 3 (2001).

[17] FRIZZIERO, E., GULMINI, M., LELLI, F., MARON, G., OH, A., ORLANDO, S., PETRUCCI, A., SQUIZZATO, S., AND TRALDI, S. Instrument Element: A New Grid component that Enables the Control of Remote Instrumentation. *CCGgrid 2* (2006).

[18] GLATARD, T., AND MONTAGNAT, J. Implementation of Turing machines with the Scufl data-flow language. In *8th IEEE International Symposium on Cluster Computing and the Grid, 2008. CCGRID'08* (2008).

[19] HASTINGS, S., OSTER, S., LANGELLA, S., ERVIN, D., KURC, T., AND SALTZ, J. Introduce: an open source toolkit for rapid development of strongly typed Grid services. *Journal of Grid Computing 5*, 4 (2007).

[20] INTERNETNZ (INTERNET NEW ZEALAND INC). Honeypot Project Website. http://www.internetnz.net.nz/workstreams/honeypot [Accessed 31 March 2009].

[21] KOMISARCZUK, P., AND KOUDRIN, A. Effect of Rerouting on NGN VoIP Quality. In *ATNAC 2007, the Australasian Telecommunication Networks and Applications Conference* (2007).

[22] KOMISARCZUK, P., SEIFERT, C., PEMBERTON, D., AND WELCH, I. Grid Enabled Internet Instruments. In *IEEE Global Telecommunications Conference, 2007. GLOBECOM'07* (2007).

[23] LAURE, E., FISHER, S., FROHNER, A., GRANDI, C., KUNSZT, P., KRENEK, A., MULMO, O., PACINI, F., PRELZ, F., WHITE, J.,

ET AL. Programming the Grid with gLite. *Computational Methods in Science and Technology 12*, 1 (2006).

[24] McGough, A., and Colling, D. The GRIDCC Project the GRIDCC Collaboration. *Communication System Software and Middleware, 2006. Comsware 2006. First International Conference on* (2006).

[25] Moestl, T., and Rombouts, P. The pdnsd Website. http://www.phys.uu.nl/~rombouts/pdnsd.html [Accessed 31 March 2009].

[26] Moore, D., Shannon, C., Voelker, G., and Savage, S. Network telescopes: Technical report. *CAIDA, April* (2004).

[27] OASIS (Organization for the Advancement of Structured Information Standards). Web Services Business Process Execution Language Version 2.0 specification. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html [Accessed 31 March 2009].

[28] Oinn, T., Addis, M., Ferris, J., Marvin, D., Senger, M., Greenwood, M., Carver, T., Glover, K., Pocock, M., Wipat, A., et al. Taverna: a tool for the composition and enactment of bioinformatics workflows, 2004.

[29] Pemberton, D., Komisarczuk, P., and Welch, I. Internet Background Radiation Arrival Density and Network Telescope Sampling Strategies. In *ATNAC 2007, the Australasian Telecommunication Networks and Applications Conference* (2007).

[30] Provos, N. Honeyd-a virtual honeypot daemon. In *10th DFN-CERT Workshop, Hamburg, Germany* (2003), vol. 2.

[31] Rajasekar, A., Wan, M., Moore, R., Schroeder, W., Kremenek, G., Jagatheesan, A., Cowart, C., Zhu, B., Chen,

S., and Olschanowsky, R. Storage resource broker-managing distributed data in a grid. *Computer Society of India Journal, special issue on SAN 33*, 4 (2003).

[32] Seifert, C., Delwadia, V., Komisarczuk, P., Stirling, D., and Welch, I. Measurement Study on Malicious Web Serversin the .nz Domain. In *Accepted to the 14th Australasian Conference on Information Security and Privacy. ACISP 2009* (2009).

[33] Seifert, C., Endicott-Popovsky, B., Frincke, D., and Komisarczuk, P. Justifying the Need for Forensically Ready Protocols: A Case Study of Identifying Malicious Web Servers Using Client Honeypots. In *4th Annual IFIP WG 11.9 International Conference on Digital Forensics, Kyoto, Japan* (2008).

[34] Seifert, C., Komisarczuk, P., and Welch, I. Identification of Malicious Web Pages with Static Heuristics. *Austalasian Telecommunication Networks and Applications Conference, Adelaide* (2008).

[35] Seifert, C., and Steenson, R. Capture-HPC Website. https://projects.honeynet.org/capture-hpc/ [Accessed 31 March 2009].

[36] Seifert, C., Steenson, R., Holz, T., Yuan, B., and Davis, M. A. Know Your Enemy: Malicious Web Servers. Tech. rep., The Honeynet Project, August 2007.

[37] Seifert, C., Welch, I., and Komisarczuk, P. HoneyC - The Low-Interaction Client Honeypot. *Proceedings of the 2007 NZCSRCS, Waikato University, Hamilton, New Zealand* (2007).

[38] Seifert, C., Welch, I., and Komisarczuk, P. Application of divide-and-conquer algorithm paradigm to improve the detection speed of high interaction client honeypots. In *The 23rd Annual ACM Symposium on Applied Computing, Brazil* (2008).

[39] Slominski, A., and von Laszewski, G. Scientific workflows survey, October 2005. http://www.extreme.indiana.edu/swf-survey/ [Accessed 31 March 2009].

[40] Slomiski, A. On using BPEL extensibility to implement OGSI and WSRF Grid workflows. *Concurr. Comput. : Pract. Exper. 18*, 10 (2006).

[41] Stirling, D., Welch, I., and Komisarczuk, P. Designing Workflows for Grid Enabled Internet Instruments. In *8th IEEE International Symposium on Cluster Computing and the Grid, 2008. CCGRID'08* (2008).

[42] Stirling, D., Welch, I., Komisarczuk, P., and Seifert, C. In *Accepted to the 9th IEEE International Symposium on Cluster Computing and the Grid, 2009. CCGRID'09.*

[43] Tieman, B. Experiences with gRAVI. In *NSF Expedition Workshop. The Role of Cyberinfrastructure in Scientific Knowledge: Emergence, Validation, and Peer Review* (2008).

[44] Wang, K. MITRE Honeyclient Project Website. http://www.honeyclient.org/ [Accessed 31 March 2009].

[45] Wang, Y.-M., Beck, D., Jiang, X., Roussev, R., Verbowski, C., Chen, S., and Kin, S. Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites That Exploit Browser Vulnerabilities. In *Proceedings of the 13th Annual Network and Distributed System Security Symposium (NDSS'06)* (2006).

[46] Welch, V., Siebenlist, F., Foster, I., Bresnahan, J., Czajkowski, K., Gawor, J., Kesselman, C., Meder, S., Pearlman, L., and Tuecke, S. Security for Grid Services. In *12th IEEE International Symposium on High Performance Distributed Computing, 2003. Proceedings* (2003).

[47] Wessels, D., Rousskov, A., Nordstrom, H., and Chadd, A. Squid Web proxy cache Website. http://www.squid-cache.org/ [Accessed 31 March 2009].

[48] World Wide Web Consortium (W3C). SOAP Version 1.2 specification. http://www.w3.org/TR/soap12-part1 [Accessed 31 March 2009].

[49] Yu, J., and Buyya, R. A Taxonomy of Workflow Management Systems for Grid Computing. *Journal of Grid Computing 3*, 3 (2005).