EDITH COWAN UNIVERSITY

CSI3344 DISTRIBUTED SYSTEMS

REPORT

# Distributed Systems

PITRE: Three-Tiered RMI Implementation

*Authors*

Mark JAMSEK (10502496)

Muzhi TIAN (10471011)

*Supervisor*

Dr Jitian XIAO

**Author Note**

Word count: 3368 (excluding listings, figures, tables, and references)

Due: 29 May 2023

Submitted: 27 May 2023

## Executive Summary

PITRE[1] demonstrates a remote method invocation (RMI) implementation of a three-tiered distributed system. All components are written in Java with a MySQL database backend. Both the second- and third-tier servers are designed to run remotely to the client, and the database itself can be hosted anywhere provided it is reachable by the database server. The primary function of the system is to generate tax return estimates and persist income records to the database. This is accomplished by allowing users to enter gross annual income in the form of twenty-six biweekly net pay and tax withheld values on the client application. These data are then sent to the server, which performs two tasks: (1) the data are packaged into a request that is sent to the database server to be converted into entities that can be persisted to the database; and (2) the server runs the data through an equation to determine whether tax monies are outstanding or have been paid in excess such that a tax return is due. The server crafts a response object, which is sent back to the client, and the result is displayed to the user.

The system is machine-independent, with Java the only runtime dependency of the client and servers, and the database is a MySQL server running in a Google Cloud instance; however, as mentioned, the database can be provisioned on any host reachable from the third-tier database server. The entire distributed system comprises three tiers: The first tier is a text-based user interface that provides the command driven client for end users to enter income records. The records are sent to the second tier, which is a server responsible for validating users, processing records, interfacing with the database server, evaluating data, and generating tax return estimates. Lastly, the database server functions as the third tier, which persists user data entered at the client application and received via requests from the second tier server to save, retrieve, and delete records. PITRE is fast, portable, and robust with an intuitive user interface that provides a friendly user experience.

[1] https://cvs.bsdbox.org/pitre

# Contents

# 1 Introduction

Personal Income Tax Return Estimator (PITRE) is a three-tiered distributed system. It is implemented in Java using remote method invocation (RMI) and MySQL Server. The primary use case PITRE aims to facilitate is generating tax return estimates for users who have provided personal income records. To realise this goal, the system was designed with modularity, portability, speed, and the user experience in mind. For this reason, Java was an obvious choice as it can run anywhere in the Java virtual machine without compromising speed, and is highly modular. Three discrete packages comprising two server libraries and a client application were implemented. The client-side application is the first tier, which is responsible for accepting and performing preliminary preprocessing of user input. These data are sent to the second-tier server via RMI, which is where users are authenticated and further, more rigorous data validation is performed. Finally, the database server acts as the third-tier, which is the interface to the MySQL database and enables persisting user data for subsequent retrieval.

This report will comprise four parts: first, a brief explanation of the problem domain and requirements of the system; second, an exposition of remote method invocation (RMI) and related techniques used in this project; third, an overview detailing the design and implementation of the PITRE distributed system; and fourth, a short manual including screenshots, example test cases, and basic instructions for endusers to follow.

## 2 Requirements and Scope

As a minimum viable product (MVP), the PITRE client application is limited in scope to the tax return estimate (TRE) client; that is, a payroll-tax client (PTC) that enables companies to calculate payroll tax is outside the scope of this project.

**Table 1:** Tax Rate

| **Gross Income** | **Tax Rate** |
|---|---|
| \$0–\$18,200 | nil |
| \$18,201–\$45,000 | 19¢ for each \$1 over \$18,200 |
| \$45,001–\$120,000 | \$5,092 plus 32.5¢ for each \$1 over \$45,000 |
| \$120,001–\$180,000 | \$29,467 plus 37¢ for each \$1 over \$120,000 |
| $>$ \$180,000 | \$51,667 plus 45¢ for each \$1 over \$180,000 |

The primary use case is to calculate and display TREs according to the provided annual income based on the tax rates listed in Table 1. In addition, a medicare levy of 2% is unconditionally applied, and depending on the private health insured status of the taxee, an additional medicare levy surcharge (MLS) may also be applied (Table 2). Records are persisted to the database where they can be retrieved or updated for subsequent TREs, which are calculated using the following equation:

$$estimate = gross\_income - net\_income - tax - medicare\_levy - medicare\_surcharge$$

If *estimate* is positive, this is the amount of money the taxee can expect to be returned from the Government's Tax Office (GTO). If negative, however, $|estimate|$ is the amount the taxee still owes in taxes to the GTO. Importantly, as a distributed system, each service shall be able to run on different machines. Nevertheless, for the purposes of this demonstration, first-, second-, and third-tier services will run on the same machine; the SQL database, however, will run on a remote Google Cloud instance running MySQL Server 5.6.

**Table 2:** Medicare Levy Surcharge

| **Gross Income** | **Surcharge** |
|---|---|
| \$0–\$90,000 | 0% |
| \$90,001–\$105,000 | 1% of gross income |
| \$105,001–\$140,000 | 1.25% of gross income |
| $>$ \$140,000 | 1.5% of gross income |

### 2.1 Client: Presentation Tier

The client will prompt the user for their personal ID, tax file number (TFN), and private health insurance status. If a TFN is provided, the client will query the server for any records that may be in the database which, if found, will be returned to the client and the user prompted to either discard or request an estimate based on these records. However, if no records are found, or a TFN is not provided, the client will prompt the user to enter twenty-six biweekly income records. As such, the functional requirements of the client include:

1. Only authenticated users shall use the system.
2. Users may or may not have a tax file number (TFN).
3. A TFN must be nine (9) numeric digits.
4. Users must provide health insurance status to request a TRE.
5. Users must enter twenty-six (26) biweekly income records to request a TRE.
6. Income records must be in the form of a tuple comprising net pay and tax withheld; that is, `net,tax`.
7. Net and tax values must be integers or numeric values that scale to two decimal places.
8. Users shall choose whether to save their records to the database.
9. Users shall choose whether to delete their records from the database.
10. Users shall be able to discard entered records and quit the application at any time.
11. Users shall be able to discard entered records and restart the application at any time.

### 2.2 Server: Business Logic Tier

In this three-tiered distributed system, the server also acts as a client, consuming the application programming interface (API) implemented by the third-tier database server. Its responsibilities as a server, however, are to authenticate users, validate tax records, and evaluate user-supplied data to generate TREs. As mentioned, the server also interfaces with the databse server by sending requests regarding data persisted to the database. For example, after validating income records, the server will issue a request to the database

server to create a new record for the user, which comprises all of the submitted income records. Similarly, after validating a TFN submitted by the client, the server will send a query to the database server for any records associated with the provided TFN. To realise these responsibilities, functional requirements of the server include:

1. Only authenticated users shall access their own records.
2. Biweekly records shall be displayed as a sequence of net pay and tax withheld tuples.
3. *Accurate* TREs according to the abovementioned formula and tax rates (Table 1 and Table 2) shall be provided to users.
4. Each TRE shall display the annual gross and net income, total tax withheld, and either tax still owed or monies to be returned.
5. Users who provide a TFN shall be able to retrieve their records from the database and request a TRE.
6. Users who provide a TFN shall be able to delete their records from, and enter new records into, the database.

### 2.3 Database Server: Data Management Tier

The database server is responsible for interfacing with the MySQL database and providing an API for the second-tier server. Its primary function is to manage requests destined for the database, such as persisting or retrieving tax-related user records. As such, functional requirements of the third-tier server include:

1. Records shall be persisted to the database.
2. Records shall be retrieved from the database.
3. Records shall be deleted from the database.
4. One set of annual income records per personal ID and TFN shall be allowed.

## 3 Remote Method Invocation

As PITRE is implemented in Java, it makes use of remote method invocation (RMI) to send messages between the client and server processes. For this reason, a brief exposition of RMI is provided. Remote method invocation (RMI) enables communication between objects running on distributed architecture. According to Tanenbaum and Steen (2014, p. 478), RMI is very similar to the remote procedure call (RPC), except that it operates on distributed objects with systemwide references. Further, such global referencing improves access transparency and parameter-passing semantics when compared to RPC. This comports with Coulouris et al. (2014, p. 220), who explain that, while closely related to RPC, RMI enables passing parameters by object reference—not only by value. On the other hand, at the lowest level, RMI is essentially the same as RPC, and both are used for the same reason: to pass messages and call remote procedures on remote hosts in distributed systems. The primary difference is that RMI is embedded in the Java programming language, while RPC is language-agnostic. When RMI was introduced by Sun

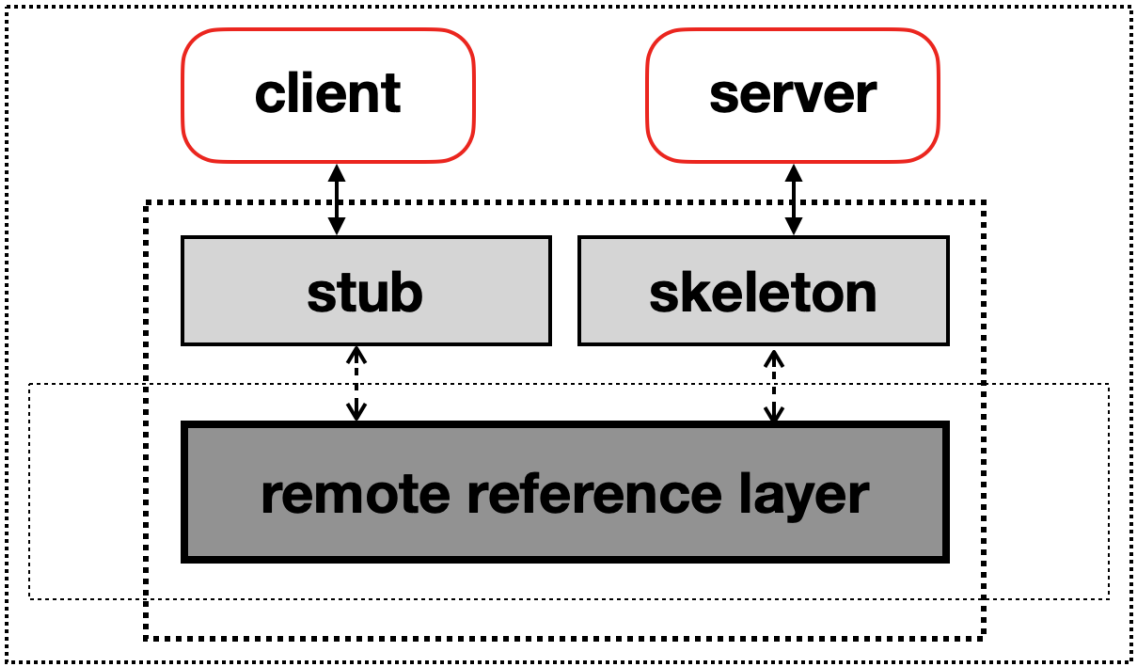


**Figure 1:** Logical illustration of the remote method invocation.

in 1997 with the release of Java Development Kit 1.1, RPC had already been around for more than a decade (Waldo, 1998, p. 5). Request for Comment (RFC) 674 established the procedure call protocol in the 1970s, which laid the theoretical underpinnings of modern

RPC protocols. Subsequent work culminated in Birrell and Nelson's Cedar RPC in 1984, commonly considered one of the earliest practical RPC implementations, and the standard on which later APIs were based (McCaffrey & Meiklejohn, 2016). There are various forms of RPCs such as synchronous, where the client blocks till a response is received, and asynchronous where the client continues working immediately after dispatching the call. Sun Java RMI is inherently synchronous, but various libraries have been implemented to overlay RMI and provide asynchronous operation.

The elegance of both methods, however, is that they abstract away the implementation details of intermachine communication. In other words, it allows clients to call procedures on remote machines as though they were local; that is, without considering the inherent complexity of sending and receiving messages. As illustrated in Figure 1, when the client invokes a call to a remote object, it is first received by what is referred to as the *stub* on the local machine. A stub is a proxy for the server-side procedure or method. The stub serialises the call and parameters, which is the process of flattening the object into a transmissible format; in the case of RMI, this is a binary-encoded stream of bytes (i.e., message). This technique is often referred to as *marshalling* (Coulouris et al., 2014, p. 178; Sayar et al., 2023, p. 25). Once the message is crafted, it is passed to the remote reference layer (RRL), which provides the low-level interface between client and server (Domenici et al., 2000). The RRL calls the `invoke()` method on the client, and sends the message to the server, where the server-side RRL receives the request and forwards the message to the *skeleton*. The skeleton is the server equivalent of the client stub, and, like the stub, is responsible for deserialising the message, which is commonly referred to as *unmarshalling*. This is the process of reconstructing the bytes back into the same call and parameters initially received at the client stub. The server can then invoke the method, whereby this process is repeated in reverse when the server sends the result back to the client.

The observant reader will ask, *how does the client know where to send the message?* This is managed by the RMI *registry*: a database of remote objects and corresponding addresses. When a object is intended for RMI, it is registered with the registry by invoking the `bind()` method, which assigns an addressable name to each object. After this, clients can lookup available objects by contacting the registry, which uses the unique name as a global reference to route RMI requests. As such, from the perspective of the programmer and readers of the source code, RMI appears indistinguishable from typical method calls.

# 4 PITRE: Overview

PITRE is a three-tier distributed system that consists of a client, also called the presentation tier; a server, often called the business logic or application tier; and database server, which is also known as the data management tier. The architecture of this design is illustrated in (Figure 2). Six Java archive (JAR) packages (Table 3) comprise the PITRE system, three of which represent each tier: the first tier is contained in the `pitre-client` package; the second tier is contained in the `pitre-server` package; and the third tier is contained in the `pitre-dbsrv` package. The fourth and fifth JARs are `pitre-interfaces` and `pitre-core`, respectively. Both of which are dependencies of each tier, and the sixth and final JAR is `pitre-dbsrv-hibernate`—a dependency of the data management tier, `pitre-dbsrv`.

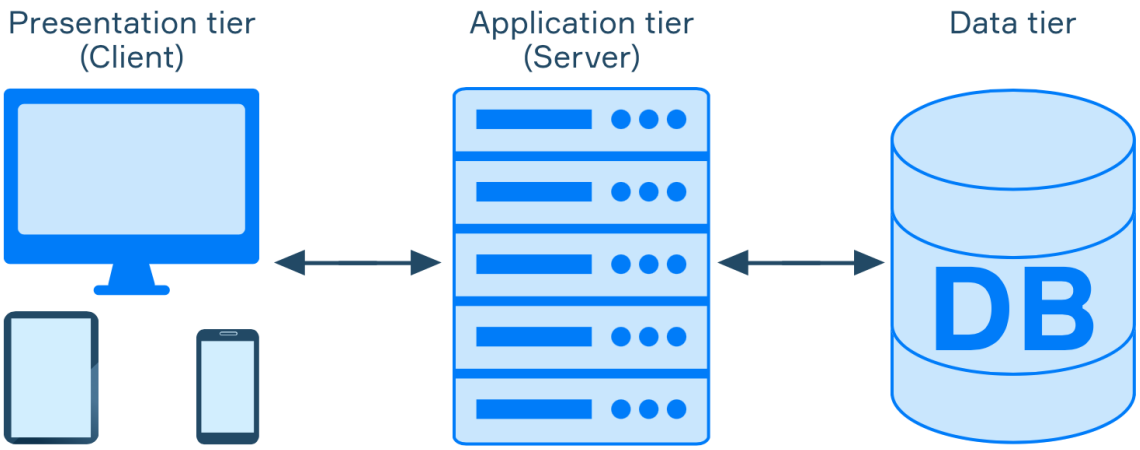


**Figure 2:** From 'Three-tier architecture' by Hyperskill, n.d. Three tier distributed system comprising the presentation tier (client), business logic tier (server), and data management tier (database server).

While the following sections explain the design and implementation details of the PITRE distributed system, one critical component is provided by the Java runtime environment (JRE): the `rmiregistry` service. This is responsible for binding names to global references of the remote objects implemented in this project, and which are usually provisioned on the remote hosts, so that each RMI is able to locate the remote object.

**Table 3:** Java Archives

| JAR | Classes | Interfaces | Dependencies |
|---|---|---|---|
| pitre-client | `CommandlineInterface`, `ConsoleInput` | `IUserInput` | pitre-{core,interfaces} |
| pitre-server | `PITREDriver`, `Auth` | `IEstimate`, `IAuth` | pitre-{core,interfaces} |
| pitre-dbsrv | `DBsrv` | `IPersist`, `IRepository`, `Serializable` | pitre-{core,interfaces}, pitre-hibernate |
| pitre-core | `Names`, `Record`, `Request`, `Result`, `Taxee`, `Taxes` | `IRecord`, `IRequest`, `ITaxee` | pitre-interfaces |
| pitre-dbsrv-hibernate | `Repository`, `DBRecord`, `DBRecordID` | `IPersist`, `IRepository` | pitre-{core,interfaces}, hibernate, mysql |
| pitre-interfaces | | `IAuth`, `IEstimate`, `IPersist`, `IRecord`, `IRequest`, `ITaxee`, `Remote`, `Serializable` | |

## 4.1 Design and Implementation

### 4.1.1 Interfaces

PITRE implements a number of interfaces that provide common abstractions to each tier. Some interfaces are used by multiple processes, while others are purely for client consumption. Table 4 documents some of these interfaces, explaining their role in the system. Tier-specific interfaces are detailed in their respective sections.

### 4.1.2 Client

The first tier hosts the client application, which provides the frontend user interface (UI) and performs preliminary preprocessing of user input before sending requests to the server. The UI is a text-based interface, implemented with the `IUserInput` interface (Table 5), which is command driven by a set of operations that are presented for user selection. At startup, the user is prompted for credentials, which are sent to the server for authentication; all clients must be authenticated before further input is accepted. This authentication request is performed by invoking the remote `IAuth` interface `login()` API. If successfully authenticated, the user will be prompted to input their personal ID before

**Table 4:** Interfaces

| Interface | Description | API |
|---|---|---|
| `IAuth` | client authentication | `login()` |
| `IRecord` | represent taxee biweekly income records | `rid()`, `net()`, `tax()`, `gross()` |
| `IRequest` | handle passing requests for taxee records | `pid()`, `taxrecord()` |
| `ITaxee` | represent a taxee including all relevant details | `id()`, `payrecords()`, `insured()`, `get()`, `add()`, `nrecords()`, `iscomplete()` |
| `IRepository` | direct communication between db server and MySQL database | `saveOrUpdate()`, `delete()`, `get()` |

being asked if they possess a tax file number, which, if in possession, they will be prompted to enter. The ID and TFN are sent to the server to be validated before the server issues a request to the database server for any associated records. If the user does not have a TFN or no records are found for the TFN provided, the user will be prompted to enter twenty-six biweekly income records. The client performs preliminary preprocessing of these records to ensure the format is valid, which is performed by the `parsetuple()` method; an additional validation layer, however is handled by the server to ensure only valid records are persisted. When all records are entered, if the server returns a response indicating the record was valid, the client will prompt the user to save and request a TRE with these records. At this point, the user can either discard and re-enter new income records or request an estimate. In the former case, the input loop will begin a new iteration, in the latter, the server public API `estimate()` is consumed and the results are displayed to the user. For a user-friendly experience, the input handler monitors user input for quit and restart events, which can be invoked at any time with the case-insensitive `Q` or `R` commands.

**Table 5:** `IUserInput` Interface

| Method | Description |
|---|---|
| `read()` | read arbitrary user input till a newline (i.e., `\n` is reached) |
| `read_username()` | like `read()`, but prompts for a username |
| `read_passwd()` | prompt for password but do not echo characters entered to standard output |

### 4.1.3 Server

The business logic tier hosts the server, which is the backend engine in the PITRE distributed system. As mentioned, this service operates as a server to the client, but also as a client to the database server. In the first instance, two interfaces present the public API to the client: `IAuth`, which is responsible for authenticating clients with the `login()` API; and `IEstimate`, which holds all of the business logic to process records and compute TREs. The public APIs provided by the `IEstimate` interface are documented in Table 6. It is the `estimate()` method, which when invoked by the client, performs the primary

**Table 6:** Server API

| Method | Description |
|---|---|
| `estimate` | calculate and return TRE result |
| `save` | save taxee income records |
| `load` | load taxee income records |
| `delete` | delete taxee income records |

*Note*. Second tier server implements the `IEstimate` interface.

use case of generating a TRE by calling `Taxes.apply()` on a `Taxee` instance. Listing 4.1 shows the `apply()` method, which returns the `Result` object that is used to display the TRE to the user. The `Result` object is an abstract data type comprising all the ingredients

needed for the server to send an estimate outcome back to the client.

```
public static Result
apply(ITaxee taxrecord)
{
        double gross, net, tax, income_tax, medicare_tax, tre;
        Result.Outcome r;

        net = total_net(taxrecord.payrecords().values().stream());
        tax = total_tax(taxrecord.payrecords().values().stream());

        gross = net + tax;

        income_tax = income_tax(gross);
        medicare_tax = medicare_tax(gross, !taxrecord.insured());

        tre = gross - net - income_tax - medicare_tax;

        if (tre > 0.0)
                r = Result.Outcome.RETURN_DUE;
        else if (tre < 0.0)
                r = Result.Outcome.TAXES_DUE;
        else  /* even money */
                r = Result.Outcome.TAXES_PAID;

        return new Result(r, Math.abs(tre), gross, net, income_tax,
            medicare_tax);
}
```

**Listing 4.1:** The `apply()` method of the `Taxes` class, which calculates a `Taxee` TRE.

In its intermediary role as a communication conduit between the presentation and data tier, the server consumes public APIs provisioned by the database server to request, persist, and delete data. This is a unique component of three-tier architecture where the

server is also a client.

### 4.1.4 Database Server

The third tier operates as the direct link to the database and as such is responsible for persisting and managing user data. In this capacity, the composite `DBsrv` class implements the `IPersist` interface, which provides three public APIs: (1) `put()`, which accepts a request that is converted into appropriate entities for persisting to the database; (2) `get()`, which accepts an ID and TFN to lookup and return as a request object after converting database entities accordingly; and (3) `delete()`, which also accepts an ID and TFN albeit to lookup and delete the entry from the database. Technically, data management tiers are also clients insofar as they consume database server APIs, often by implementing methods or calling library routines that correspond to SQL statements such as *INSERT* and *UPDATE*. In PITRE, this is managed by the `IRepository` interface (Table 4) implemented by the `Repository` class—a component of `DBsrv`—that overrides some, and calls other, methods of the Java `hibernate` object-relational mapping (ORM) library—a popular framework for mapping Java data types to SQL. For the purpose of this demonstration, the `DBsrv` process runs on the same host as the `Srv` process, but the MySQL database is running on a remote Google Cloud instance in a datacentre somewhere in Sydney. Figure 3 expands on the model illustrated in Figure 2 to more accurately demonstrate this architecture.

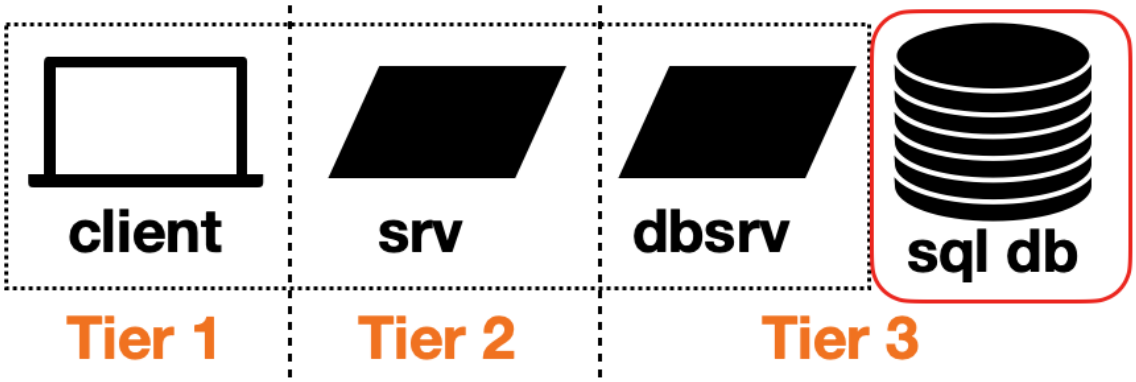


**Figure 3:** Actual topology of the PITRE distributed system. The third tier comprises the database server module, which runs locally with the client and server in the demonstration, and the MySQL Server database, which is in a Google Cloud instance in a Sydney datacentre.

# 5 User Manual

PITRE can be built and launched with a single command invoked from the root of the work tree: `pitre.sh` and `pitre.ps1` on Unix-like and Windows platforms, respectively. However, build and run tasks can be performed independently (Listing 5.1).

```
pitre % ./pitre.sh -h
pitre.sh 0.10 - PITRE distributed system build and launch utility

      -B        do not build PITRE
      -h        show help and exit
      -R        do not run PITRE
      -V        show version number and exit
      -v        increase output verbosity

usage: pitre.sh [-BhRVv]
```

**Listing 5.1:** `pitre.sh -h` invocation to display help and usage.

## 5.1 Build

In both the repository and source tarball, the POSIX compliant shell script `pitre.sh` is provided for Unix-like systems. This script should be invoked with the `-R` flag in the root of the work tree to automate the build process. Messages are written to standard output at each stage of the build and logs are written to the `log/` directory.

```
pitre % ./pitre.sh -R
[+] checking java exists...ok
[+] checking gradle exists...ok
[+] building pitre packages...ok
[+] build finished
run PITRE with './pitre.sh -B'
```

**Listing 5.2:** Example of a successful `pitre.sh -R` invocation to build the PITRE distributed system.

Listing 5.2 demonstrates output from a typical build. The only build dependencies are Java and Gradle, and the latter is provided in the repository and source tarball. The PowerShell script `pitre.ps1` is provided for Windows platforms, which is functionally equivalent to the POSIX script.

## 5.2 Run

A successful build operation must have been completed before running PITRE. Then, `pitre.sh` should be invoked with the `-B` option to bypass the build process, initialise all processes, and start the client application. An example `pitre.sh -B` invocation is shown in Listing 5.3. On line 8, the `rmiregistry` is daemonised followed by a brief wait to allow the registry to initialise. Without the wait, servers will be unable to bind their references so the system will fail to properly start. On lines 10 and 11, the server and database server, respectively, are initialised—again with a preset wait interval—before the client is launched on line 12. After this, authentication prompts for a username on line 19.

```
1  pitre % ./pitre.sh -B
2  [+] checking java exists...ok
3  [+] pitre remote interfaces: ./pitre-interfaces/build/libs/pitre-interfaces-0.10.jar -> ok
4  [+] pitre sqldb server: ./pitre-dbsrv/build/libs/pitre-dbsrv-0.10.jar -> ok
5  [+] pitre server: ./pitre-server/build/libs/pitre-server-0.10.jar -> ok
6  [+] pitre client: ./pitre-client/build/libs/pitre-client-0.10.jar -> ok
7  [+] adding remote interfaces to classpath...ok
8  [+] daemonise rmiregistry
9  [+] wait for rmiregistry to initialise...ok
10 [+] initialising pitre db server...
11 [+] initialising pitre server...
12 [+] initialising pitre client...
13 PITRE - Personal Income Tax Return Estimate
14 ----------------------------------------
15
16 Please follow the prompts
17 (q)uit or (r)estart at any time
18
19 username:
```

**Listing 5.3:** Abridged output from a successful `pitre.sh -B` invocation.[2]

A short screencast of the build process followed by a brief runtime demonstration of PITRE can be viewed at the following link: https://ss.jamsek.net/pitre-demo.mp4

[2]Initialisation output from both servers has been trimmed from the listing for the sake of clarity.

### 5.2.1 Authentication

Credentials for the demonstration release:

**username:** `test`
**password:** `test`

## 5.3 Test Cases

PITRE has robust input handling to ensure only legal syntax and valid records are accepted for generating TREs. Example test cases below demonstrate the defensive programming employed during development.

### 5.3.1 Authentication

Listing 5.4 illustrates input handling of various username and password credentials.

```
PITRE - Personal Income Tax Return Estimate
-------------------------------------------

Please follow the prompts
(q)uit or (r)estart at any time

username:
username must not be empty
username: test
characters will not be echoed to screen
password:
<srv> password must not be empty
please enter a valid username and password
username: test
password:
<srv> login failed
please enter a valid username and password
```

**Listing 5.4:** Failed authentication: empty username; empty password; and invalid credentials

### 5.3.2 Tax File Number Validation

Listing 5.5 illustrates validation of several invalid TFN formats including illegal characters and incorrect number of digits.

```
tax file number:
>>
please provide input
tax file number:
>> A01234567
tfn must be 9 digits
tax file number:
>> 01234567
tfn must be 9 digits
tax file number:
>> 0123456789
tfn must be 9 digits
```

**Listing 5.5:** Invalid tax file number: empty input, illegal character; and incorrect number of digits.

### 5.3.3 Income Record Validation

Listing 5.6 illustrates validation of illegal input to ensure income records are entered in the required `net,tax` tuple format.

```
enter 26 biweekly pay records in the form 'net_pay,tax_withtheld'
(e.g., 4318.45,2027.70)
(q)uit or (r)estart at any time to discard changes
net,tax record 01:
>> six thousand, twenty-five hundred
values must be integers or floats up to 2 decimal places
net,tax record 01:
>> 6000 2500
invalid format; please enter records as 'net,tax' tuples
net,tax record 01:
>>
```

**Listing 5.6:** Example invalid string and whitespace delimited tuple input for income record 01.

### 5.3.4 No Tax File Number

Listing 5.7 demonstrates the case of a user with no TFN.

```
do you have a tfn [y/N]:
>> n
enter 26 biweekly pay records in the form 'net_pay,tax_withtheld'
(e.g., 4318.45,2027.70)
(q)uit or (r)estart at any time to discard changes
net,tax record 01:
>>
```

**Listing 5.7:** Demonstration of a user with no tax file number.

### 5.3.5 Tax File Number Found

Listing 5.8 demonstrates the case of a user with a TFN found in the database.

```
do you have a tfn [y/N]:
>> y
tax file number:
>> 123456789
<srv> load user records from database: jamsek,123456789
<sql> retrieve 26 rows from the database
tax records found: jamsek 123456789
 1:   5150.10   2527.70
 2:   5150.10   2527.70
...
25:   5150.10   2527.70
26:   5150.10   2527.70
request (e)stimate, (d)iscard and enter new records, (q)uit [d/E/q]:
>>
```

**Listing 5.8:** Example demonstration of a user with a tax file number on record.

### 5.3.6 Tax File Number Not Found

Listing 5.9 demonstrates the case of a user with a TFN not found in the database.

```
do you have a tfn [y/N]:
>> y
tax file number:
>> 012345678
<srv> load user records from database: demo1,012345678
<sql> retrieve 0 rows from the database
tax records not found: demo1 012345678
enter 26 biweekly pay records in the form 'net_pay,tax_withtheld'
(e.g., 4318.45,2027.70)
(q)uit or (r)estart at any time to discard changes
net,tax record 01:
>>
```

**Listing 5.9:** Example demonstration of a user with a tax file number not on record.

## 5.4 Windows Usage

Listing 5.10 illustrates `pitre.ps1` usage in Windows PowerShell.

```
PS C:\src\pitre> .\pitre.ps1 -help
pitre.ps1 0.10 - PITRE distributed system build and launch utility

      -buildonly        do not run PITRE
      -help             show help and exit
      -runonly          do not build PITRE
      -verbose          increase output verbosity
      -version          show version number and exit

usage: pitre.ps1 [-buildonly  -help  -runonly  -version  -verbose]
```

**Listing 5.10:** `pitre.ps1 -help` output.

# 6 Summary

As the report demonstrated, this project explored distributed systems through the design and development of a three-tier RMI implementation in Java. The fact RMI was invented by Sun and introduced in Java, made the language a compelling choice. And the large, mature ecosystem made implementation trivial—especially when compared to previous and continuing work on the Game of Trees[3] (`got`) and Fossil[4] distributed version control systems written with sockets in ANSI C. Correspondingly, RMI and extensive library support facilitated rapid development; indeed, third-tier work was relatively straight forward with the hibernate ORM solution. In addition, the "*write once, run anywhere*" cross-platform benefit of Java compiled bytecode—without compromising speed—was an especially appealing factor. However, one notable limitation was considered during the initial design phase: RMI in Java exacts a monolithic architecure. By implementing PITRE in Java, the project would be *limited* to Java; that is, any desired extensions to the feature set would have to be implemented in Java. For example, new server implementations would be confined to Java due to object serialization compatibility. Likewise, development of a web frontend client would have to be written as a Java applet. This is compounded by Java being highly susceptible to remote code execution because of deserialisation vulnerabilities that can be used to chain return-oriented programming (ROP) gadgets (Sayar et al., 2023, p. 37). In fact, according to van der Stock et al. (2021), this class of vulnerability is ranked eighth on the Open Worldwide Application Security Project (OWASP) Top Ten list of the most dangerous web application security vulnerabilities. Trivial development notwithstanding, due to these factors, neither RMI nor Java would be recommended for future distributed programming intended to be published. Instead, gRPC[5] in C++ offers several advantages such as protocol buffers for serialisation, native asynchronous communication, and support for multiple languages. Nonetheless, PITRE demonstrates the convenience of RMI in distributed systems development and presented a learning opportunity for distributed programming with a once favoured communication model in Java.

[3] https://gameoftrees.org
[4] https://fossil-scm.org
[5] https://grpc.io

## References

Australian Taxation Office. (2021). Single touch payroll. https://www.ato.gov.au/Business/Single-Touch-Payroll/

Coulouris, G., Dollimore, J., Kindberg, T. & Blair, G. (2014). *Distributed Systems Concepts and Design* (5th ed.). Addison-Wesley.

Domenici, A., Superiore, S., Anna, P. & E-mail, I. (2000). Object-oriented techniques for distributed computation. https://www.researchgate.net/publication/2367494_Object-Oriented_Techniques_for_Distributed_Computation

Hyperskill. (n.d.). *Three-tier architecture*. https://hyperskill.org/learn/step/25083

McCaffrey, C. & Meiklejohn, C. (2016). A brief history of distributed programming: RPC. *Code Mesh 2016*. https://codemesh.io/codemesh2016/caitie-mccaffrey.

Mousa, A., Karabatak, M. & Mustafa, T. (2020). Database security threats and challenges. *2020 8th International Symposium on Digital Forensics and Security (ISDFS)*, 1–5. https://doi.org/10.1109/ISDFS49300.2020.9116436

Sayar, I., Bartel, A., Bodden, E. & Le Traon, Y. (2023). An in-depth study of java deserialization remote-code execution exploits and vulnerabilities. *ACM Trans. Softw. Eng. Methodol.*, *32*(1), 25–70. https://doi.org/10.1145/3554732

Taboada, G. L., Touriño, J. & Doallo, R. (2009). Java for High Performance Computing: Assessment of Current Research and Practice. *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, 30–39. https://doi.org/10.1145/1596655.1596661

Tanenbaum, A. S. & Steen, M. V. (2014). *Distributed Systems Principles and Paradigms* (2nd ed.). Pearson.

van der Stock, A., Glas, B., Smithline, N. & Gigler, T. (2021). *Open worldwide application security project top ten* (tech. rep. OWASP Top 10:2021). The Open Worldwide Application Security Project Foundation. Wakefield, Massachusetts. https://doi.org/https://owasp.org/Top10/A08_2021-Software_and_Data_Integrity_Failures/

Waldo, J. (1998). Remote procedure calls and java remote method invocation. *IEEE Concurrency*, *6*(3), 5–7. https://doi.org/10.1109/4434.708248

Weisenburger, P., Wirth, J. & Salvaneschi, G. (2020). A survey of multitier programming. *ACM Comput. Surv.*, *53*(4). https://doi.org/10.1145/3397495