Appendix D

MASSEY UNIVERSITY
Application for Approval of Request to Embargo a Thesis
(Pursuant to AC98/168 (Revised 2), Approved by Academic Board 17/02/99)

Name of Candidate: JEFFREY PAUL BRIDGES ID Number: 12159110

Degree: MASTER OF ENGINEERING Dept/Institute/School: SEAT

Thesis title: Development of a multi-historian SCADA data collection middleware

Name of Chief Supervisor: Edmund Lai Telephone Ext: 41580

As author of the above named thesis, I request that my thesis be embargoed from public access until (date) 01/05/2015 for the following reasons:

☑ Thesis contains commercially sensitive information.

☐ Thesis contains information which is personal or private and/or which was given on the basis that it not be disclosed.

☐ Immediate disclosure of thesis contents would not allow the author a reasonable opportunity to publish all or part of the thesis.

☐ Other (specify):

Please explain here why you think this request is justified:

The thesis contains details of the implementation of a commercial product. Making this information public immediately could provide an advantage to their competitors.

Signed (Candidate): JBridges Date: 29/04/2013

Endorsed (Chief Supervisor): Name Date: 29/4/2013

Approved/Not Approved (Representative of VC): [signature] Date: 2/05/13

Note: Copies of this form, once approved by the representative of the Vice-Chancellor, must be bound into every copy of the thesis.

Development of a multi-historian SCADA data collection middleware

A thesis presented in partial fulfilment of the
requirements for the degree of

Master of Engineering
in
Computer and Electronic Engineering

at Massey University, Albany, New Zealand

Jeffrey Paul Bridges
2013

Abstract

This thesis details the implementation of a SCADA middleware package designed to provide a common interface to the most commonly used SCADA historians in use within local government in New Zealand. The middleware integrates with a New Zealand developed cloud based solution that is used throughout the country for compliance monitoring and reporting of water and wastewater treatment facilities.

The middleware connects via the internet to the historians hosted on-site within council networks and retrieves data which is replicated in a local database. This approach provides backup for the data and allows it to be accessed quickly when required for reporting. The middleware database is hosted within a Microsoft SQL Server instance to ensure compatibility and ease of rollout when paired with the commercial cloud solution.

To ensure reliable data collection and resilience to connection interruption all transactions made against the client historians are controlled by a queue stored in the middleware database. The middleware includes an inbuilt mechanism for integrity checking to ensure all data available in the client historian is collected and has been designed to run with minimal intervention of company staff.

To reduce storage requirements the middleware includes a data deduplication system that removes repeated samples from the SQL Server database after integrity checking has been performed. This provides a lossless compression mechanism that does not alter the precision of the data collected.

The middleware implementation has now been in use in a production environment collecting data from councils for compliance purposes for approximately six months. During this time data collection has been very reliable with the middleware handling numerous connection outages without intervention.

Acknowledgements

Thanks to Associate Professor Edmund Lai for providing supervision and guidance over the course of this project.

I am very grateful to Dr Peter Johnson for providing the opportunity to work on this project, and providing his support and guidance over the past year.

This project was partially funded by the Ministry of Science + Innovation. Thanks to Chris Lock for his efforts in securing this funding.

This project would not have been possible without support from the IT and utilities staff at a number of councils around New Zealand who provided access to their operational data.

A special thanks to my partner Alexandra for her support and understanding during the long hours required to complete this project.

Thanks to my family who provided support and motivation throughout the project.

Contents

List of Figures

List of Tables

Chapter 1: Introduction

This thesis details the implementation of middleware that facilitates an interface to multiple SCADA historian systems with a particular focus on the SCADA systems and historians in use within the water and wastewater treatment sections of local government in New Zealand.

The middleware is intended to integrate with a commercial cloud based compliance monitoring and reporting solution that is used throughout New Zealand by local government. The product provides reporting based on manually collected data which will be augmented with data obtained from SCADA telemetry systems with the introduction of this middleware.

1.1 Motivation

While most SCADA historian packages provide a means to perform reporting the information on which they can report is generally limited in scope as the entire information set needed for compliance is not held in the historian. In the case of the water and wastewater treatment industry the information from the SCADA system is combined with manually tested results and lab data for reporting purposes.

The cloud based compliance monitoring and reporting solution for which this middleware is designed aims to provide a seamless reporting interface which bridges all potential data sources and therefore significantly reduce the amount of manual effort required to assemble compliance reports. One important aspect to this is the ability to source data from a number of different SCADA historian implementations.

Providing councils with a solution that can report from multiple historians gives them many more options when considering technology upgrades. When building new facilities the potential for vendor lock-in will be reduced as the SCADA system and historian implemented will no longer have to match the product used at other facilities. Where existing infrastructure is being upgraded the historic data can be retained when a new historian solution is implemented, with reporting continuing seamlessly with the legacy data being reported alongside data from the newly implemented historian.

1.2 Objectives

The first objective is to determine the SCADA historian solutions most commonly used in New Zealand. The outcome will be a list of historians for which interfaces will be developed to facilitate access via the middleware.

The next objective will be the implementation of a test and development environment in which the middleware implementation can occur. Where possible this includes obtaining access to SCADA historian implementations and obtaining historic data from councils for test and benchmarking purposes.

This will be followed by the definition of a common interface that will be used by consumers of data contained in the middleware as a means to access stored data. This common interface will be integrated in the commercial cloud solution to enable reporting on data collected by the middleware.

The final objective will be to implement the middleware. This will include devising an architecture that is both reliable and resilient to connection disruptions. The middleware database will need to provide a level of performance in excess of the volume of data being produced by clients and include compression to ensure disk space is used efficiently. To demonstrate that the middleware is reliable and fit for purpose it will need significant testing in the production environment where it will form part of the compliance reporting solution for clients.

1.3 Thesis Overview

Chapter 2 provides a review of information relevant to the middleware implementation. This includes a background of SCADA systems and their corresponding historian products and methods available for accessing the data they contain, including methodologies for securing data while in transport. This is followed by a brief investigation of data collector architecture and storage concepts. Guidelines for a common interface are then presented, followed by an overview of development methodologies and conclusions.

Chapter 3 gives a detailed overview of the SCADA historians most commonly encountered during a survey of local government infrastructure. This includes details of the data storage and data access methodologies adopted by each historian.

Chapter 4 covers the existing cloud solution in to which the middleware is designed to integrate. This includes an overview of the hardware and network configurations employed in the production environment. The operating system and software image used for hosting clients is detailed, along with implementation requirements for the middleware.

Chapter 5 covers the implementation of a test and development environment including the hardware and software setup for all middleware development and testing. The common

interface used by the middleware data consumers to extract data from the middleware is then defined. This is followed by an investigation of historian data access methodologies and techniques for ensuring the reliability and resilience of data collection. An overview of potential options for securing historian access and in-transit data is provided, followed by an investigation of data compression techniques.

Chapter 6 covers the implementation of the middleware service including the development framework and solution architecture. The database schema used for the middleware database is explained in detail. This is followed by a comprehensive description of the transaction queue used to ensure reliability. The workings of the application and worker threads are detailed, along with the data integrity checking scheme. The data compression scheme implemented is described, followed by details of the access methodology and table mappings used for each target historian. The chapter ends with a brief overview of the security implemented in production and details of the middleware installation in the production environment.

Chapter 7 provides the conclusions and an overview of the milestones achieved during implementation of the middleware. This is followed by a number of suggestions for future improvements that could be implemented to improve the performance and functionality of the middleware.

Chapter 2: Literature Review

This chapter gives an overview of supervisory control and data acquisition (SCADA) systems and includes an overview of the current methods used for collecting, archiving and providing access to historic data. Potential data storage solutions for the proposed middleware are investigated along with development methodologies that may be suited to the project.

2.1 Background Information

Supervisory control and data acquisition (SCADA) systems are widely used throughout the water and wastewater sector of local government. The SCADA systems in use in these facilities are used for both data acquisition and process control allowing relatively automated operation of the water and wastewater treatment systems. The system utilizes remote terminal units (RTU) and programmable logic controllers (PLC) interconnected via a control network to implement the functionality required to operate the plant.

The RTU devices serve as an interface to the physical world providing telemetry information gathered from connected sensors including temperature monitors, pH probes, flow meters and turbidity meters. These sensors provide an analogue signal which is sampled and converted to a digital representation by the RTU. The RTU also provide a means to control external equipment through outputs that can be used to switch relays controlling high powered equipment.

Process control logic is implemented using PLCs connected to the control network. The PLCs receive telemetry data from RTU devices installed as required throughout the facility. The PLC contains software which implements the required level of automation through control of RTU outputs based on telemetry data from the facility.

Information from both the RTU and PLC devices is made available to the facility operators via a computer display using a human-machine interface (HMI). The HMI can allow operators to view raw data from connected RTUs, show the control data calculated by PLCs and allow overrides of control settings if required. The system running the HMI generally includes a database system containing a log of SCADA tag data. A single SCADA tag can represent either a physical value obtained from an RTU device or a calculated value generated by a PLC. The database contains a log of time-value pairs for each tag and allows the HMI system to perform trending based on historic data.

To facilitate long-term centralised storage of data contained in tag databases there are a number of historian products available. The historian connects to the HMI system and can retrieve and

store both real-time and historic data. This data is stored on the historian system using a number of different strategies which are investigated in more detail in the next section. The proprietary and differing nature of these historian systems can make data extraction difficult and is why the common interface described in this thesis has been proposed.

2.2 Theory

This section investigates the theory involved with individual components of the proposed solution.

2.2.1 SCADA Historians

SCADA historians use a variety of database technologies for data storage ranging from commercial relational databases such as Microsoft SQL Server and Oracle to high performance proprietary database systems that may be required for fast real-time data access and high-volume data storage (Zolotová, Flochová, & Ocelíková, 2005). The advantages and disadvantages of the relational database when compared with a proprietary solution are covered in this section.

Utilizing a proprietary database can provide high rates of data storage with some solutions able to store hundreds of thousands of samples per second when using the appropriate hardware and configuration (GE Intelligent Platforms). This compares favourably with relational database systems which on the appropriate hardware can provide write speeds in the order of approximately ten thousand samples per second (Schnieder Electric Industries SAS, 2011). Using a hybrid model with relatively static data stored in a relational database and high volume time series data stored in a proprietary database can also provide the high write speeds of a fully proprietary solution (Middleton, 2011).

The data storage requirement for a historian varies based on the database system used. When storing uncompressed data a typical relational database requires 40 bytes per sample, while proprietary solutions can store the same sample using only 6 bytes of storage (GE Intelligent Platforms). In both the relational and proprietary case compression can be used to reduce data storage requirements. A common strategy for compression relies on the use of a dead band which prevents new samples being committed to storage until the current value has changed more than a given amount when compared to the last stored value (Vasyutynskyy, 2007). In a test of 400,000 samples stored with a 1% dead band GE Intelligent Platforms found the data storage required was reduced by 78%.

2.2.2 Historian Data Access

A crucial part of historian functionality is to facilitate the retrieval of stored data for later use. Most historians provide a number of different means to access the data they contain; these are detailed in this section.

For historians utilizing a Microsoft SQL Server database access is available via a number of methods, the two more common being the SQL Server Native Client and the SQL Server Managed Provider. The SQL Server Native Client provides both an Object Linking and Embedding, Database (OLE DB) driver and an Open Database Connectivity (ODBC) driver for accessing data stored within the historian SQL database.

The OLE DB driver provides a Component Object Model (COM) application programming interface (API) which allows applications written for the Microsoft Windows platform to execute queries against the SQL server. As COM is language neutral these queries can be made from applications written in a variety of programming languages.

ODBC is a standard C API for accessing databases. The SQL Server ODBC driver can be used by applications written in C, C++ and Visual Basic (Microsoft Corporation). Applications written to utilize the ODBC driver communicate with the SQL Server using C function calls with the driver passing SQL statements to the server and returning the results to the application.

The SQL Server Managed Provider is an ADO.NET data provider that can be used by applications written for the .NET Common Language Runtime, including those written using C#, VB.NET and other .NET languages. ADO.NET data providers implement a consistent interface for access to a variety of data sources, including SQL Server (Microsoft Corporation). The SQL Server ADO.NET provider allows commands to be executed directly against the server while also allowing the Language-Integrated Query model to be used to create objects representing the data model of the relational database (Microsoft Corporation).

SQL Server is also accessible from other operating systems such as Linux using tools such as FreeTDS. This tool allows native access to data stored within SQL databases from UNIX and Linux based platforms (FreeTDS Project, 2011). FreeTDS consists of C libraries that implement the tabular data stream protocol used for communication with SQL Server.

Historians utilizing proprietary database storage provide access to stored data using similar data access drivers. In some cases an OLE DB driver is provided (GE Intelligent Platforms) which implements the standard interfaces required for querying the historian database. Despite the

interface to the driver being standard the language and structure of the queries required to extract data can vary between historian implementations.

Data access to some historians is also provided via an open connectivity historical data access (OPC HDA) API. OPC HDA describes a standard interface for retrieving data from applications that store historical data. An OPC HDA client can connect and retrieve data from any OPC HDA compliant server with the OPC HDA API (OPC Foundation, 2013). The server is able to both provide raw and interpreted data as requested by the client.

Proprietary APIs for data access are also provided by the manufacturers of some historians (GE Intelligent Platforms). These APIs require code or libraries provided by the manufacturer to be included in the application code to facilitate access to the data (GE Fanuc Automation, 2006). These APIs can be manufacturer specific preventing application data access code being used across historians from different manufacturers.

2.2.3 Data Transport and Security

To access data from SCADA historians a connection over TCP/IP is used (GE Intelligent Platforms). As TCP/IP encompasses the protocols used for internet communication it is possible for historian data to be accessed remotely via the internet. The protocol at the application layer varies between historians based on the data access methodologies implemented by the manufacturers.

As the historian data may travel over the public internet security can be required to limit access to only desired users. For historians utilizing Microsoft SQL Server for data storage the security scheme provided within SQL Server can be used to assign permissions on a per-user basis. Roles are assigned per user and can limit the data the user can read and write on a per database table basis. This can prevent private data being read or data being overwritten. Requiring a login for all users also prevents unauthorised access to the database.

Historians utilizing a proprietary database can rely on other security mechanisms for data protection. If the historian makes data available via OPC HDA the authentication scheme contained within the OPC HDA specification can prevent unauthorized access to the content of the database with permissions able to be specified on a per-user basis to prevent read or write of specified data (Mahnke, Leitner, & Damm, 2009).

Where an OLE DB provider is used for data access the authentication scheme must be implemented by the historian manufacturer and varies between historians. In most historian implementations this authentication is implemented as a username and password combination.

To provide additional security a firewall can be used to restrict access to the historian computer. Using firewall rules access can be restricted by IP address to only those computers allowed to access historian data. This adds an additional layer of security that helps prevent data access in the event that authentication details are compromised as requests from computers that are not in the allow list are ignored. Firewalling also helps prevent brute force attacks and attempts to compromise the security model using flaws in application code.

When data travels across the public internet there can be no guarantee that the transmission is private. To prevent snooping of data in transit an encryption scheme is required. Microsoft SQL Server provides encryption through the optional use of Secure Sockets Layer (Microsoft Corporation). SSL encryption can be made either optional or mandatory for connecting clients with either a 40-bit or 128-bit level of encryption available. The client-server authentication process is always encrypted to prevent credentials being stolen.

The OPC HDA specification allows for a secure channel to be used for communications. The encryption algorithm used is negotiated when a session is established between historian and client (Mahnke, Leitner, & Damm, 2009).

For OLE DB providers any encryption must be implemented by the manufacturer in the proprietary protocol used for communication between historian and client.

2.2.4 Data Collector Architecture

As the data collector implemented for this middleware will be running and continually collecting data a high degree of reliability and resilience is required. As the data may be collected via unreliable links the collector needs the ability to track data that may have been missed and populate this data when the link is restored. To further ensure the validity and completeness of collected data a method for integrity checking is also required.

Existing research regarding the architecture of such an application proved difficult to locate so a substantial amount of development and testing in this area will be required to ensure a robust solution.

2.2.5 Data Storage

Once data has been collected from the historian a system for storing the data locally is required. This role is typically performed by a relational database with entities representing the data needing to be stored. Relationships can be made using foreign keys to link between attributes in database tables.

Data is added to and retrieved from the database using Structured Query Language (SQL). Some languages allow SQL tables and their relationships to be represented as entities in code. C# provides LINQ to SQL which allows the traditional stream of records queried from an SQL Server to be represented as objects which can be interacted with using standard object-orientated programming techniques (Troelsen, 2007).

To improve the speed of queries it is essential a proper index be used. The database table index is much like that of a book and provides a quick means of finding a particular piece of data. Without an index the entire database would need to be scanned until the desired data is located (Allen, 2004).

SQL Server databases can contain a large amount of data with the number of rows being limited only by the available storage (Microsoft Corporation, n.d.). The tables can contain up to 1000 unique indexes to improve data retrieval performance.

2.2.6 Common Interface

An application programming interface facilitates interaction between software components. The interface can facilitate the sharing of information between two separate applications and can include specifications for tasks and data structures. Developing a good interface is crucial as incorrect design decisions can be magnified as a single interface is often reused by numerous applications.

A good application programming interface will facilitate simple development of a client application without additional code needed to manage operations performed using the interface. To help ensure a good interface design the interface needs to be designed with the ease of use of the calling application well considered. Self-documenting input and output parameters can make use from a client application easier.

Thoroughly documenting the interface before beginning development can help provide an interface that is not encumbered by the host application architecture and ensure the interface meets the needs of calling applications.

To facilitate software using the common interface to access samples from multiple SCADA historians the collected data needs to be presented in a consistent form. This could take the form of a common interface that only exposes features common to all historians, or an interface with the ability to report the attributes and features supported by the selected data source.

While SCADA historians perform a similar function the data stored by individual products does vary. The basic information stored by all historians for a given sample incudes the date and time at which the sample was taken and the value of the sample at that time. Some historians store additional information with each sample. This additional metadata can include sample quality information, time zone data and the sampling mode used to obtain the data (GE Fanuc Automation, 2006).

2.2.7 Development Methodologies

There are a number of different programming methodologies that aim to bring structure and discipline to the software development process. This section discusses the traditional heavyweight waterfall approach in comparison to the more lightweight iterative and agile development methodologies.

Waterfall development relies on heavy planning and project management to provide a repeatable, predefined liner development process. The development of code should not be started until all requirements have been determined with requirements being frozen early in the project. Project management techniques such as Gantt charts and milestones should be used through the project lifecycle (Edberg, Ivanova, & Kuechler, 2012). A typical project lifecycle for a project utilizing the waterfall method from the planning to deployment stage could last anywhere from three months to two years (Shore & Warden, 2008) and is shown in Figure 1.

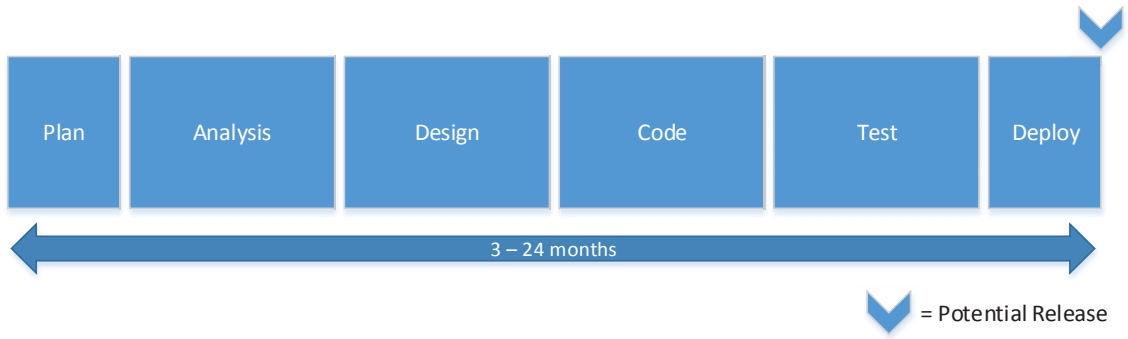

Figure 1: The Waterfall method (Shore & Warden, 2008)

The iterative method has many similarities when compared to the waterfall method with the same linear timeline consisting of planning, analysis, design, development, testing and deployment. The iterative condenses this linear timeline into a much shorter period of a one to three month cycle which is repeated over the length of the project; this allows for more frequent releases of software for deployment (Shore & Warden, 2008). An example of the iterative method is shown in Figure 2.

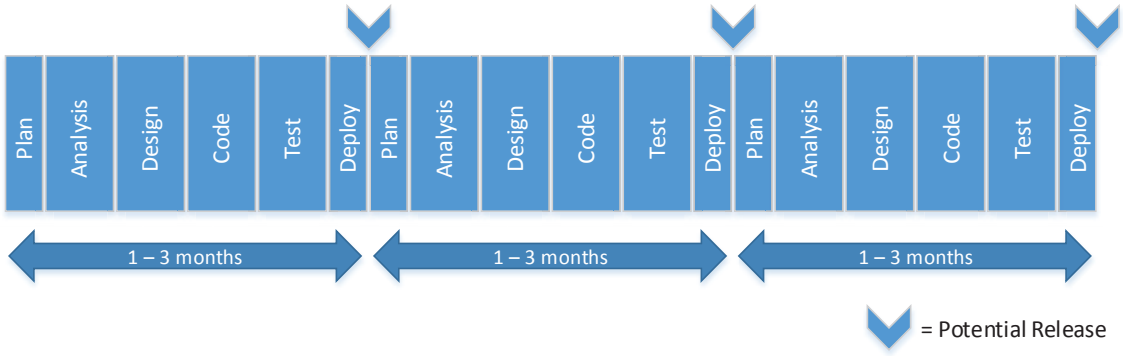

Figure 2: The Iterative method (Shore & Warden, 2008)

The agile methodologies provide a much more flexible means of development that is able to accommodate and easily respond to change. Time that would often be devoted to planning is instead spent developing with requirements being continually determined and refined as the project proceeds (Edberg, Ivanova, & Kuechler, 2012). The lifecycle of an agile project is divided into small blocks of approximately a week long with the aim of delivering software that is able to be deployed internally or with customers at the end of each block. After a small amount of planning the analysis, design, coding and testing occur simultaneously building toward the deployment at the end of the week. This approach allows plans to change quickly with new features and changes made available in deployed software within a much shorter timeframe than would be the case with waterfall or iterative development (Shore & Warden, 2008). This approach is shown in Figure 3.

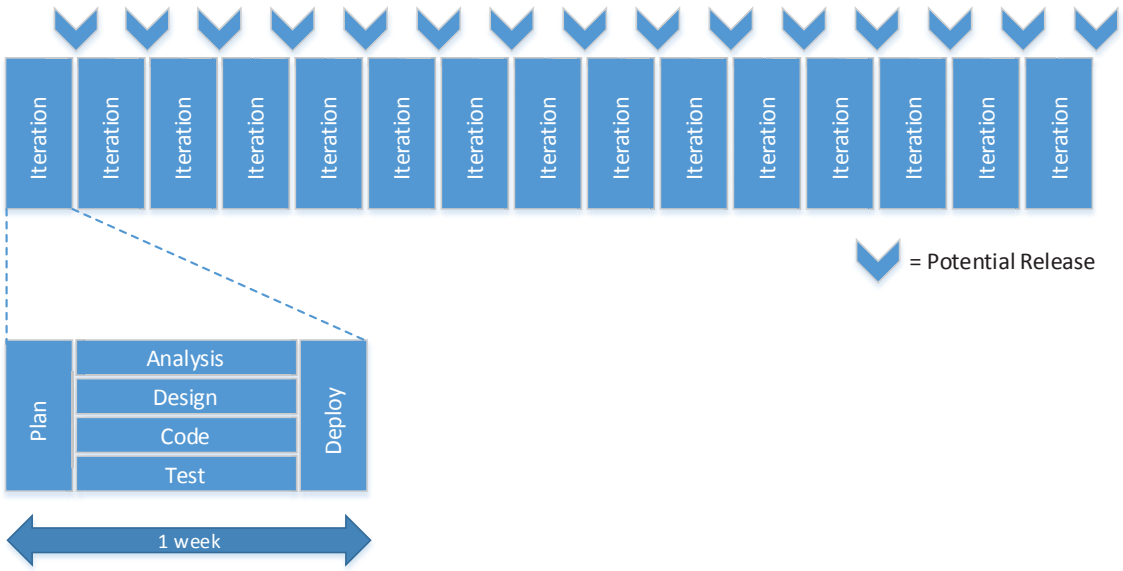

Figure 3: Agile development (Shore & Warden, 2008)

By definition there are no strict methods by which to perform agile development as it has no concrete practices to follow and may involve an approach that continually changes based on the situation at hand (Shore & Warden, 2008).

2.3 Summary

This chapter has covered a number of components which will be interfaced via the proposed middleware solution. While there are a number of SCADA historian implementations with each generally using a different data storage strategy there are interfaces provided to give access to the stored data. As this data may need to be transferred over the public internet there are methods to secure this data and restrict access to a list of source addresses.

Building a reliable, resilient data collector is essential and will require substantial research and testing as existing research isn't readily available. The data collector will need the ability to collect data that may have been missed due to connectivity or other failures.

A relational database such as that offered by SQL Server can provide a large amount of data storage accessible across a number of different platforms. Utilizing correct indexes on the tables within the database is essential to ensure decent performance.

Defining a good interface for access to data provided via the middleware is crucial. Once an application programming interface is deployed and integrated into other applications making changes becomes very difficult and a poor interface can make developing downstream software considerably more difficult.

Through the use of agile programming methodologies software releases can be produced for testing at regular intervals. The agile approach also allows plans to be quickly changed with new features being introduced throughout the development process being made available quickly in test releases.

Chapter 3: SCADA Historians

This section provides an overview of historians commonly used in the New Zealand Local Government water and wastewater treatment sector. The historian products detailed in this section were selected as they were the most commonly encountered historians in a survey of local government SCADA historian implementations.

3.1 Citect Historian

Citect Historian is produced by Schneider Electric and runs within a Microsoft Windows environment. The historian supports collecting real-time data from Citect SCADA and other open connectivity data access (OPC DA) compliant data sources and can retrieve historic data from trends on Citect SCADA systems. Citect Historian uses the Microsoft SQL Server relational database for storage of collected data and dependent on the historian hardware configuration can store up to 10,000 changes per second into the database.

Each piece of collected data is stored with a timestamp and quality flag. Storing this along with the sample value requires 40 bytes of storage for each data point. To reduce the amount of storage required the historian implements a user configurable deadband which prevents new values being written to the database unless the current value is greater or less than the last stored value plus or minus the deadband amount. This prevents small and insignificant changes causing ripples in the stored data. The deadband is configurable per SCADA tag.

As Citect Historian uses a Microsoft SQL Server database for data storage a number of methods are available to access the stored data. These methods include native access via ODBC and OLE DB drivers, and managed access using a .NET data access provider. Queries are written using T-SQL, a Microsoft proprietary extension to structured query language (SQL). The historian also provides a web interface and plugin for Excel, both of which allow data extraction.

3.2 GE Proficy Historian

Proficy Historian is produced by GE Intelligent Platforms, again for Microsoft Windows platforms. Proficy utilizes individual data collectors implemented for a number of different sources including Proficy HMI/SCADA systems and OPC compliant data sources. The data collectors implement a store and forward architecture to prevent data loss caused by lost connectivity between the historian and data collector.

Proficy uses a proprietary data storage solution allowing hundreds of thousands of data points per second to be stored. Raw data can be stored uncompressed with a data size of 6 bytes per

data point. Data storage requirements can be reduced through the use of a per-tag sampling deadband to filter small insignificant changes to the data value being saved.

As the Proficy Historian database is of a proprietary format a Proficy specific component can be required to integrate data from the historian with another system. For this purpose Proficy includes an OLE DB driver and open connectivity historical data access (OPC HDA) interface. OLE DB drivers are discussed in more detail in the next section.

3.3 Invensys Wonderware Historian

Invensys Wonderware Historian is a high-performance historian product that supports up to 500,000 SCADA tags per historian. It runs in a Microsoft Windows environment and is implemented as a time-series extension to Microsoft SQL Server. Implementing the historian in this manner gives the performance and compression advantages of a custom storage format while providing still providing an easy, non-proprietary means of accessing data. In the case of Wonderware Historian the data is accessed via SQL Server through a standard T-SQL implementation with some additional query syntax for accessing historian specific functionality.

Microsoft SQL Server has a variety of options available for accessing data, including native access via ODBC and OLE DB drivers, and using the .NET data access provider. This allows Wonderware Historian to be readily accessed from external applications without having any proprietary software or drivers installed on the connecting machine.

3.4 QTech DATRAN

DATRAN is a SCADA solution produced in New Zealand by QTech Data Systems. The DATRAN solution is used across New Zealand in the water & wastewater industry, including in a number of local councils.

The DATRAN solution uses an installation of Microsoft SQL Server to store collected data. This results in an access methodology similar to other historians utilizing Microsoft SQL Server, being via ODBC drivers, OLE DB drivers or a .NET data access provider.

3.5 Hilltop DataTamer

Hilltop DataTamer is a suite of products produced in New Zealand by Hilltop Software. The product suite is widely used amongst local and regional councils within New Zealand. Hilltop DataTamer uses a proprietary format for data storage which will reportedly support read speeds of 100,000 rows per second. To facilitate access to this proprietary format stored data a web based module is available.

The web data access module is implemented as a native extension for Internet Information Services 7.5 or higher which requires a minimum operating system of Windows Server 2008 R2 x64 or Windows 7 x64. The module allows requests for site listings, which details sites available to request data from. A list of measurements available for each site can then be requested, from which data for a specified date range can then be obtained.

The responses to requests are sent as XML over the HTTP connection made to the module. Utilizing HTTP allows for the connection to be encrypted using HTTPS to prevent the data being intercepted in transit, and also allows authentication to provide a reasonable level of access control and security for the data. An HTTP server also provides a small potential attack surface as the database server containing the data does not need to be directly exposed to the internet. HTTP also provides an inbuilt mechanism for compression to reduce the data transfer bandwidth requirement.

Chapter 4: Existing Cloud Solution

The middleware detailed in this document has been designed to integrate with a New Zealand developed off the shelf product that is used within local and regional government. The software is web-based and hosted for clients from commercial datacentres around the country to provide redundancy and data security. The product is primarily used for compliance monitoring of water and wastewater treatment plants and performs reporting based on manually derived and telemetry sourced data. The middleware is required to integrate with the existing production environment without additional software where possible to simplify deployment to new and existing virtual machines. Compatibility with the existing hardware and virtual machine environment is essential to ensure the middleware can be used with existing infrastructure.

4.1 Production Hardware

The existing production hardware consists of a number of 1U rack-mount servers in commercial datacentres around the country. While the specifications of the machines vary based on age the following list gives an approximate overview of a typical production machine:

- 1U rack-mount server
- Quad core processor
- 24+ gigabytes memory
- Hardware based RAID controller
- Multiple hard disks in a RAID array giving 1-2 terabytes of redundant storage
- Dual gigabit Ethernet interfaces

These machines have been operating reliably for a number of years without significant downtime or hardware failures. As the service has grown additional machines have been added to allow the service to scale.

4.2 Production Software

The product runs primarily atop Microsoft Windows and Microsoft SQL Server. Each client instance is allocated a virtual machine in the primary datacentre on which their data is held and web portal is hosted. These virtual instances are deployed from a base image containing a Windows installation with SQL Server preinstalled. The following is a list of software loaded in the production image:

- Microsoft Windows Server 2008 R2 x64
- Microsoft SQL Server 2008 R2 Standard x64
- Internet Information Services 7.5 running in integrated pipeline mode

- Microsoft .NET Framework 4.0 with ASP.NET 4.0
- Web portal application code

Updates to the Microsoft products installed as part of the image are handled via a Windows Server Update Services 3.0 SP2 installation which is configured as the update source. Updates to the web portal code are performed by a custom developed background service which polls an internal server containing MSI packages with updated web portal and background service code. Both the Microsoft updates and web portal updates are installed automatically during a weekly maintenance window at a suitable time contractually agreed with their clients.

4.3 Production Virtual Machine Software

The production virtual machine instances are hosted atop a VMware ESXi v4.1.0 hypervisor, managed by an instance of VMware vSphere Client. New client virtual machines are deployed using VMware vCenter convertor.

Utilizing virtual machines for client instances provides the flexibility to move client virtual machine instances easily between servers to balance load as necessary. It also provides redundancy and failover capabilities to unsure uptime requirements are met.

4.4 Production Network

Connectivity in the production environment is provided by two separate networks. An internal network connects all of the client virtual machines hosted on the VMware ESXi server. To facilitate connectivity to the internet another virtual machine is deployed on each ESXi server. This virtual machine runs Debian Linux and has a network interface connected to both the internal network with the client virtual machines and the external network that is connected to the internet.

When web requests are made for client virtual machines the HTTP or HTTPS traffic first arrives at the Debian Linux virtual machine. The HTTP or HTTPS request is processed by the open-source web server and proxy nginx. To determine the correct client virtual machine for which the request is intended nginx examines the host headers of the HTTP or HTTPS request. The request is then modified so the response is sent back to nginx, which in turn is forwarded back via the internet to the source of the original request.

This approach allows the client virtual machines to be relatively isolated from the internet, with no network interfaces connected to the external network. The Debian Linux machine has only a small number of ports open to requests from the internet, giving a smaller attack surface. As

client virtual machines are moved between ESXi server machines communication can be continued without DNS changes as the client virtual machine retains the same IP address on the internal network.

To facilitate client virtual machines having access to the internet the Debian Linux virtual machine also performs network address translation via iptables. The network address translation process modifies the source and destination of packets as required to route traffic from the client virtual machines to the internet and return responses to the origin.

4.5 Production Schematic

To provide an overview of the production infrastructure including hardware, virtual machines and network configuration a schematic is shown below.

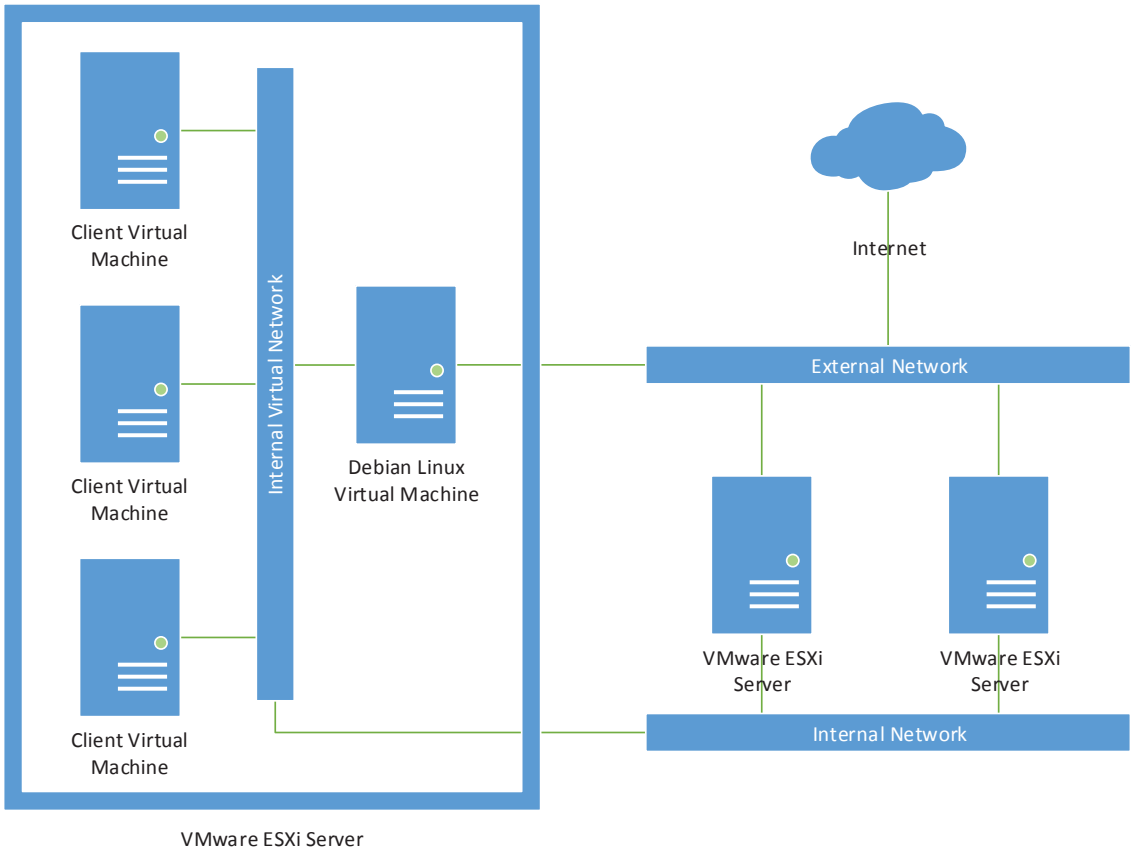

Figure 4: Production environment schematic

4.6 Implementation Requirements

To ensure an easy roll out of the middleware software changes and additions must be kept to a minimum. Any additional software required may need to be manually deployed to each existing client virtual machine, which could amount to a considerable time commitment for staff.

Additional software will also have resource overheads which may affect the responsiveness of the web portal and resource allocation across the virtual machine pool.

Any additional software installed would require a new methodology for deployment of security patches and bug fixes. While this is achieved simply for the Microsoft products currently installed in the image via Windows Server Update Services a generic solution for products from other vendors is not part of the client image. It is imperative to deploy security patches as quickly as possible for internet facing servers to avoid compromise of client data.

The commercial solution is provided on a monthly subscription basis to clients, with the software licence costs being integrated as part of this monthly fee. Additional costs of licencing further software may not be agreeable to clients or the solution provider. This provides further incentive to develop a middleware implementation based on the existing Microsoft technologies that are already used in production and are part of the virtual machine image.

The contracts with production clients specifies that for disaster recovery purposes a copy of the SQL database containing their data will be transferred to a location they designate on a daily basis. This is supplemented periodically by an installer containing the web portal and associated applications which can be installed on a local server by the client should extended downtime occur in the production environment. The middleware implementation must be installable by clients on a server or virtual machine with specifications and software similar to that of the production virtual machines should the need arise.

Chapter 5: Investigation and Planning

This section details work performed during the investigation and planning stage of the project. This encompassed building a self-contained test and development environment with a number of virtual machine instances used for emulation of the production cloud solution and client historians.

After provisioning the development hardware a common interface was planned which when used in production will facilitate access to the middleware implementation. Investigation of potential solutions for aspects of the middleware implementation were then performed to ensure when developed the middleware would achieve the required specifications.

5.1 Test Environment Hardware

To facilitate development and testing of the SCADA middleware a test environment was required. Ensuring compatibility with the production hardware and software was essential, so therefore the test environment used hardware that was as similar as practicably available. To facilitate connection to remote historians a 10 Mb fibre-based internet connection with multiple fixed IP address was used.

5.1.1 Hardware

The hardware provided consisted of a Supermicro 1U rack mount server with specifications as listed below.

- Supermicro SuperServer 5016T-TB 1U rack mount server
- Intel Core i7-950 Quad-core CPU at 3.06 GHz
- 24 GB DDR3 1333 MHz non-ECC memory
- Intel SASWT4I SAS/SATA RAID controller
- 2x Seagate ST31000333AS 1TB hard disks in RAID1 redundant array
- 2x Intel 82547L Gigabit Ethernet controllers

The server was mounted in a rack with other development machines and was very similar in specification to the production servers on which the middleware will be used.

As multiple virtual machines ran on the server a reasonable amount of RAM was required. The 24GB installed in the server allowed approximately 4GB per virtual machine to be allocated as is described in the following software section.

To allow for a high level of availability and data security a RAID1 mirrored array was used. In a RAID1 setup an identical copy of the stored data and application software is stored on each of

the two hard disks in the array. In the event of a disk failure the machine continues to run as normal on a single hard disk until the failed disk is replaced, at which point the array is rebuilt from the still-functional drive. This setup proved invaluable during development and testing as one of the drives in the machine failed. A replacement drive was sourced while the failed drive was sent for replacement under warranty which ensured the RAID array was restored to an optimal state as quickly as possible. There was no downtime, data loss or other interruptions caused by the drive failure. The Intel hardware RAID controller was used to support this RAID array as software RAID solutions were not supported by the hypervisor used on the server.

Two Ethernet ports were available on the server. To facilitate communication with machines on the internal network one of these ports was connected to a gigabit switch shared with other machines on the local network. The other port was connected to the router/switch providing internet access.

5.2 Test Environment Software

The software installed in the test environment was identical to that used in production. To facilitate running multiple distinct machine instances and simulate the disparate nature of the historian, middleware and data consumers a hypervisor was used.

5.2.1 Hypervisor

To facilitate running multiple virtual machine instances on the test environment server a hypervisor was installed. Two options were considered for the hypervisor – the first option being a hosted hypervisor package that ran atop an existing operating system installation. This approach results in a larger amount of abstraction between the hypervisor and the underlying hardware as all access to the server hardware is managed by the operating system hosting the hypervisor. The hosting operating system also requires memory and additional system resources, meaning less resource is available to the virtual machines running atop the hypervisor.

The second option was a native bare-metal hypervisor that ran on the hardware directly. Without the abstraction of another underlying operating system the hypervisor is free to use and manage system resources as needed. The bare-metal hypervisor has a comparatively low memory requirement and greatly reduced disk space footprint when compared to a hosted hypervisor. Hardware support can be an issue when using a bare-metal hypervisor as it cannot rely on operating system drivers for communication with hardware. The bare-metal hypervisor must contain driver support for all of the hardware in the server on which it will run so will generally run on a more restricted set of hardware.

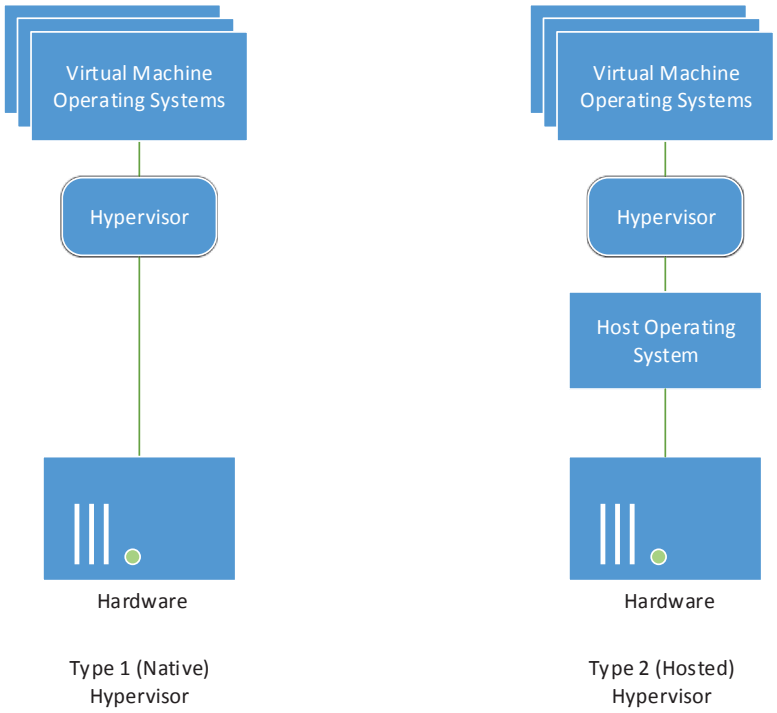

Figure 5: Hypervisor type comparison

A bare-metal hypervisor was chosen as with a smaller memory and CPU overhead it makes more resource available to the virtual machines it controls. Both VMware ESXi and Microsoft Hyper-V were considered as compatible options. Each provided support for the full range of hardware in the Supermicro server including the hardware RAID controller and network interfaces. The management tools provided with both hypervisors were suitable and provided a simple means for deploying new virtual machines and monitoring the server hardware. Both hypervisors were available in free editions which provided ample functionality for the test environment.

Ultimately VMware ESXi was chosen as the preferred hypervisor. The primary factor in making this decision was integration with the hardware health monitoring system used in the production environment. This meant the test environment hardware could be monitored alongside the production hardware with email alerting should any failures occur. The version of the hypervisor used was VMware ESXi 4.1.0 as to maintain commonality with the production environment. VMware vSphere Client was used for management of the virtual machines, with VMware vCenter Converter being used for deployment of virtual machines from a base image.

5.2.2 Middleware Virtual Machine

The middleware virtual machine was provisioned to run test versions of the middleware as it was developed. To facilitate this the following hardware was provisioned for the virtual machine:

- Dual-core processor
- 4 gigabytes RAM
- 100 gigabyte virtual disk
- 2x virtual network interfaces

To mirror the production environment as closely as possible the following software was installed:

- Windows Server 2008 R2 Standard x64 with all Windows Update patches
- SQL Server 2008 R2 Standard Edition SP1
- .NET Framework 4.0

To facilitate remote deployment and debugging of the middleware the Visual Studio 2010 remote debugger was also installed on the virtual machine. The remote debugger allowed code to be monitored and stepped-through as if it was running locally on the workstation used for development. This meant the code could be monitored without having a full set of development tools installed on the middleware virtual machine.

To provide internal network connectivity one of the virtual network interfaces was bound to the internal virtual switch which was in turn bound to an external port connected to the local LAN switch. This local switch was shared with other machines such as the development workstation.

The second virtual network interface was bound to the external virtual switch which allowed a direct connection to the router providing internet access. This virtual network interface had a fixed real-world IP address accessible from the internet and allowed connections to client SCADA historians to be made.

5.2.3 Reporting Virtual Machine

In the production environment reporting functionality is accessed by clients using a web-based interface. This web-based interface and corresponding background services generate the report using data sourced from a number of different sources including data collected manually that is populated via the web interface, and laboratory sample test results that are retrieved from laboratory systems.

The reporting virtual machine was built to run a modified copy of the production web interface which contained additional code to allow report data from SCADA systems to be sourced via the SCADA middleware. The reporting virtual machine and the middleware virtual machine both resided on the same virtual network switch to allow direct communication between machines.

The virtual hardware provisioned for the reporting virtual machine was the same as used in the production environment. This consisted of the following:

- Dual-core processor
- 2 gigabytes RAM
- 50 gigabyte virtual disk
- 2x virtual network interfaces

The software installed also mirrored the production environment and was as listed below:

- Windows Server 2008 R2 Standard x64 with all Windows Update patches
- Internet Information Services 7.5 running in integrated pipeline mode
- SQL Server 2008 R2 Standard Edition SP1
- Microsoft .NET Framework 4.0 with ASP.NET 4.0

Debugging of the changes made to the web-based reporting interface was performed on a development workstation with the ASP.NET Development Server so the remote debugging toolkit was not required.

In a similar manner to the middleware virtual machine one virtual network interface was bound to the internal network and the other bound to the external internet connection. The external connection allowed the reporting web interface to be accessible to clients via their internet connections.

5.2.4 Citect Historian Virtual Machine

To perform testing against a Citect Historian data source a virtual machine with a Citect database was provisioned. The virtual hardware specifications were as below:

- Dual-core processor
- 4 gigabytes RAM
- 100 gigabyte virtual disk
- 1x virtual network interface

The software installed on the virtual machine was as follows:

- Windows Server 2008 R2 Standard x64 with all Windows Update patches
- SQL Server 2008 R2 Standard Edition SP1

As Citect Historian uses an SQL Server database to store its configuration and SCADA data an installation of Citect Historian was not required. A backup database of approximately 12 gigabytes in size with around 10 years of historic data was obtained from a client. This database was restored in the instance of SQL Server running on the virtual machine and used as a data source during initial development and testing of the middleware.

As the database stored on this machine was static and the machine had no requirement for incoming connections from the internet only a single network interface was necessary. This interface was connected to the internal virtual switch shared with other virtual machines and the local network.

5.2.5 GE Proficy Virtual Machine

To develop and test the interface to GE Proficy Historian a virtual machine was provisioned. The specifications were as follows:

- Dual-core processor
- 4 gigabytes RAM
- 100 gigabyte virtual disk
- 1x virtual network interface

The software installed on the virtual machine was as follows:

- Windows Server 2008 R2 Standard x64 with all Windows Update patches
- GE Proficy Historian 4.0

As GE Proficy Historian uses a proprietary means to store data an installation of Proficy Historian was required. GE provide a free 25-tag edition of Proficy Historian which was installed on the virtual machine. The historian comes with a built-in simulated data provider which was configured to log samples to the historian database which were then retrieved by the middleware during development and testing. This provided a Proficy Historian data source without requiring any data to be provided by clients.

The historian had no requirement for external connections so the network interface of the virtual machine was connected to the internal virtual switch. The middleware connected via this internal interface to retrieve data from the historian.

5.2.6 Invensys Wonderware Virtual Machine

Invensys Wonderware is in use within numerous New Zealand councils, though during development of the middleware it was not possible to negotiate access to a live installation of

Wonderware. To allow development to proceed a database capable of emulating Wonderware was created and hosted on a virtual machine of the following specifications:

- Dual-core processor
- 2 gigabytes RAM
- 50 gigabyte virtual disk
- 1x virtual network interface

The virtual machine had the following software installed:

- Windows Server 2008 R2 Standard x64 with all Windows Update patches
- SQL Server 2008 R2 Standard Edition SP1

Wonderware uses the SQL Server as an interface to access stored data. To emulate an instance of Wonderware database tables were created with a similar schema and populated with data obtained from other sources.

5.2.7 QTech DATRAN Virtual Machine

To facilitate development of the DATRAN interface a backup of a dataset was obtained from a client. This was restored on a virtual machine with the following specifications:

- Dual-core processor
- 2 gigabytes RAM
- 50 gigabyte virtual disk
- 1x virtual network interface

The virtual machine had the following software installed:

- Windows Server 2008 R2 Standard x64 with all Windows Update patches
- SQL Server 2008 R2 Standard Edition SP1

DATRAN uses an SQL database to store data so the client backup dataset was restored into a database on the SQL server. Accordingly a copy of DATRAN was not required on the virtual machine. The backup dataset contained approximately six years of historic data and was approximately 2 gigabytes in size. The middleware connected to the SQL database via network connection to retrieve data.

As there was no historian installation an external internet connection was not required. Therefore the network interface on the virtual machine was connected to the virtual switch shared by the internal network.

5.2.8 Hilltop Virtual Machine

Hilltop DataTamer is in use in a number of councils within New Zealand. It was not possible to arrange access to a client Hilltop DataTamer installation during the course of development so instead a piece of software was developed to simulate the interface used by Hilltop for access to data stored in the proprietary format data files. This was run on a virtual machine of the following specifications:

- Single-core processor
- 2 gigabytes RAM
- 50 gigabyte virtual disk
- 1x virtual network interface

The following software was used in the simulation of Hilltop data:

- Windows Server 2008 R2 Standard Edition x64 with all Windows Update patches
- Internet Information Services 7.5 running in integrated pipeline mode
- .NET Framework 4.0 with ASP.NET 4.0

Hilltop uses an XML over HTTP mechanism to facilitate access to the data it stores. The simulator ran as an HTTP handler in Internet Information Services and returned randomly generated data formatted as XML in the form Hilltop DataTamer outputs. This allowed development and testing of a Hilltop interface without access to a Hilltop DataTamer installation.

As the virtual machine was only required to accept incoming HTTP connections from the middleware on the internal network a single network interface was all that was required.

5.2.9 Large Dataset Virtual Machine

During development of the middleware a large dataset consisting of approximately 60 gigabytes of data spanning 10 years was obtained from a client. This data was in the form of an SQL database that had been populated from a SCADA historian by an internally developed tool. To allow this data to be used for testing and benchmarking purposes it was restored in an SQL database on a virtual machine with specifications as follows:

- Dual-core processor

- 4 gigabytes RAM
- 100 gigabyte virtual disk
- 1x virtual network interface

The following software was used on the virtual machine:

- Windows Server 2008 R2 Standard Edition x64 with all Windows Update patches
- SQL Server 2008 R2 Standard Edition SP1

The data was made available on the internal network where it could be queried by benchmarking tools developed on and run from the development workstation. The machine therefore had no requirement for external network access and was connected only to the internal virtual network switch.

5.2.10 Development Workstation

All development of the middleware and associated test and benchmarking tools was performed on a workstation separate to the test server and virtual machines. The workstation was a desktop machine and consisted of the following hardware:

- Intel Core i7-930 Quad-core CPU at 2.8 GHz
- 12 GB DDR3 1333 MHz non-ECC memory
- 2x Seagate ST31000333AS 1TB hard disks in RAID1 redundant array
- Intel 82547L Gigabit Ethernet controllers

The software installed on the development workstation was as follows:

- Windows 7 Ultimate SP1 x64 with all Windows Update patches
- Visual Studio 2010 Ultimate Edition
- SQL Server 2008 R2 Management Tools
- .NET Framework 4.0

The development workstation was connected to the internal network using a gigabit Ethernet interface. As new versions of middleware were developed the software was deployed to the middleware virtual machine and tested using the Visual Studio remote debugger. Access to the databases running atop virtual machines in the test environment for management and benchmarking purposes was via the internal network.

5.2.11 Network Overview

The diagram in Figure 6 provides an overview of the network setup used to interface the test environment virtual machines, the development workstation and the internet connection used for accessing client data.

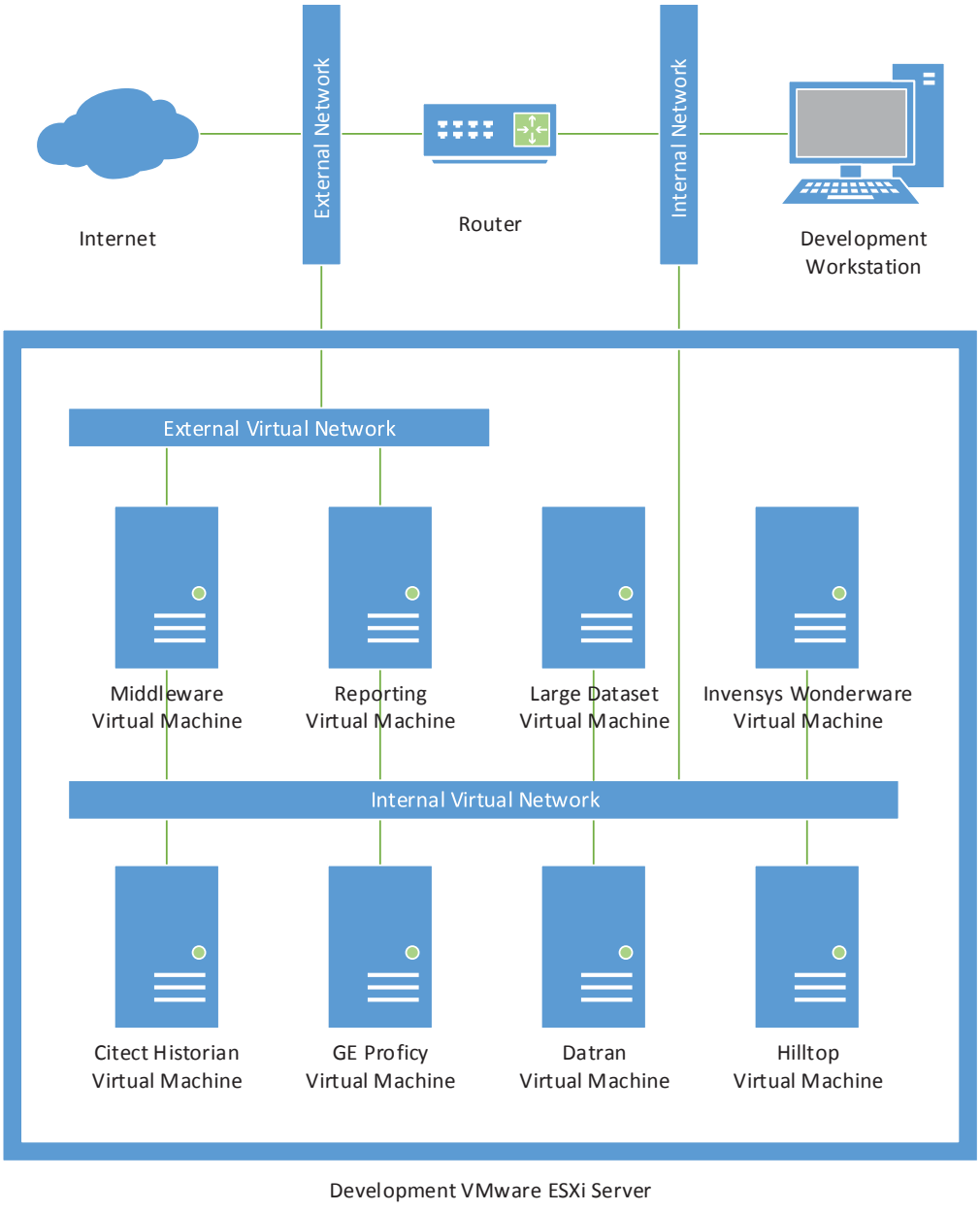

Figure 6: Development network

5.3 Common Interface

To facilitate external applications accessing data via the middleware a common interface is required. This common interface must provide a consistent set of methods for accessing data which will return results in a specific format regardless of the data source and will be implemented in a C# assembly which will be called from the production report generation code.

Being implemented in an assembly will allow the interface to be used in other applications with a minimal amount of code required to access stored data. This section details the methods and return types implemented in the common interface for data access.

5.3.1 Tag Listing

To facilitate any data access via the common interface a list of available tags needs to be obtained. The GetTags method of the common interface requires no arguments and returns a data table containing a list of all tags available via the middleware. Figure 7 shows the schema of the data table returned by GetTags.

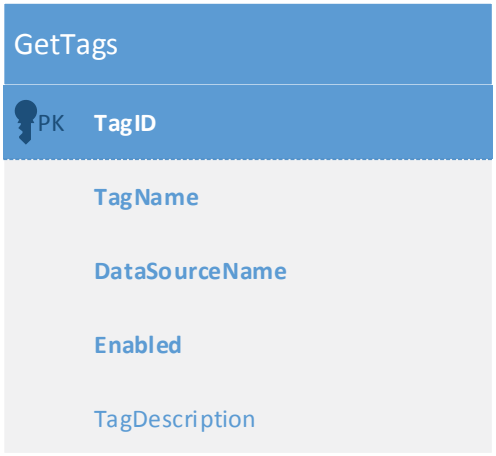

Figure 7: Common interface GetTags response data table schema

- TagID: The TagID column provides an integer ID corresponding to a given tag which is assigned as new tags are found in source historians. The ID is required for all other operations that are available via the middleware as it provides a non-ambiguous means of referencing tags. The TagID column will always be populated in the tag list.
- TagName: The TagName column is a string which provides information to aid end-users in locating the required SCADA tag. The name is derived from tag listings requested from source historians and will vary in format depending on information made available by each historian. The TagName column will always be populated in the tag list.
- DataSourceName: The DataSourceName column is a string that is made available for grouping purposes. As the tag names between historians may not always be unique the data source name provides additional information to help differentiate tags. The data source name is specified when adding a new historian to the middleware. The DataSourceName column will always be populated in the tag list.

- Enabled: The Enabled column returns a Boolean value indicating whether data collection from the source historian is currently enabled for the tag. The Enabled column will always be populated in the tag list.
- TagDescription: The TagDescription column will provide further information about the tag if possible. Depending on the source historian this could include the units of measurement, the physical location, or arbitrary text provided when the tag was added to the source historian. The TagDescription column may be NULL if the source historian is not able to provide suitable data from which to populate the column.

5.3.2 Tag Collection Status

To prevent unnecessary data being collected tags must be individually enabled for collection from their source historian. In the production web interface this is done through an association of virtual tags to one or more middleware tags. The SetCollectionStatus method takes a TagID integer and Enabled Boolean value as inputs. SetCollectionStatus will not change the collection status for any tag if an attempt is made to set the collection status for an invalid TagID. Figure 8 shows the inputs required by the SetCollectionStatus method.

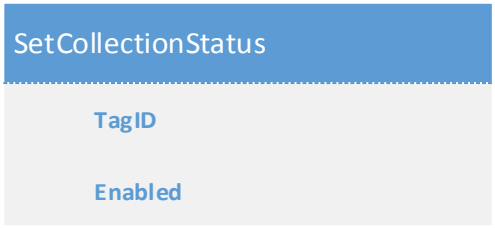

Figure 8: Common interface SetCollectionStatus method inputs

- TagID: The TagID input is an integer and corresponds to the tag ID returned by the GetTags method. The TagID input is required when setting tag collection status.
- Enabled: The Enabled input is a Boolean that specifies the desired collection status for the tag. When the Boolean value passed in is TRUE collection of data will be enabled for the tag, with FALSE disabling collection of data for the tag.

5.3.3 Tag Data Retrieval

Once a tag ID has been obtained from the tag listing and had collection enabled via the SetCollectionStatus method data will become available as it is retrieved from the source historian. The GetData method facilitates access to this data, taking a tag ID and date range as input and returning a data table containing the requested data. Figure 9 shows the inputs required by the GetData method.

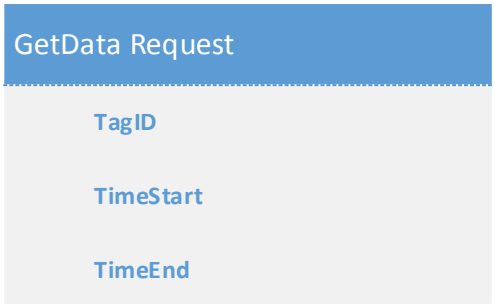

Figure 9: Common interface GetData method inputs

- TagID: The TagID input is an integer and corresponds to the tag ID returned by the GetTags method. The TagID input is required when retrieving data.
- TimeStart: The TimeStart input is a DateTime object which specifies the start of the time range for which to query data. The TimeStart input is required when retrieving data.
- TimeEnd: The TimeEnd input is a DateTime object which specifies the end of the time range for which to query data. The TimeEnd input is required when retrieving data.

The GetData method will return data for the specified tag that has a sample time greater than or equal to TimeStart and less than TimeEnd. If TimeStart or TimeEnd has a DateTimeKind of Local or Unspecified the time will be converted to UTC. GetData will return an empty table if an attempt is made to get data for an invalid TagID. The data table schema as returned by GetData is shown in Figure 10.

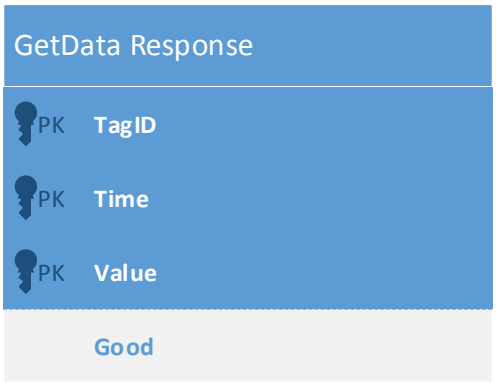

Figure 10: Common interface GetData response table schema

- TagID: The TagID column contains an integer with the ID of the tag data was requested for. The TagID column will always contain a value when returning requested data.
- Time: The Time column contains a DateTime with the time at which the sample was taken. The time is in UTC to ensure accurate timekeeping and to prevent data overlapping and being out of order after a return from daylight saving to standard time. The Time column will always contain a value.

- Value: The Value column contains the floating point value of the sample at the given time. The Value column will always contain a value, though the value may not be accurate if the Good column is FALSE.
- Good: The Good column contains a Boolean that indicates the validity of the sample value. If the Good column contains TRUE the historian considered the sample value to be reliable. If the Good column contains FALSE the historian considered the sample value invalid and therefore should not be used for reporting. The Good column will always contain either TRUE or FALSE.

5.3.4 Tag Data Backfill

In some instances data for a given tag may not have been available for collection in the source historian when requested by the middleware. In the event that this missing data is made available in the historian at a later date a method is required to request that the middleware make another attempt to collect the data. This process is known as a backfill and may be requested by calling the QueueDataBackfill method. The QueueDataBackfill method takes a tag ID and time range as shown in Figure 11.

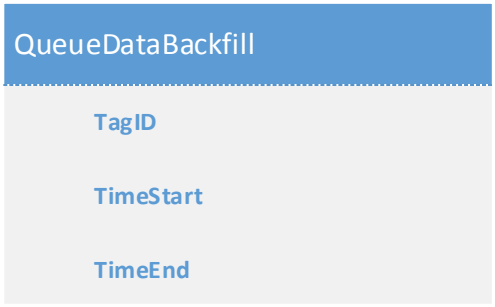

Figure 11: Common interface QueueDataBackfill method inputs

- TagID: The TagID input is an integer and corresponds to the tag ID returned by the GetTags method. The TagID input is required when backfilling data.
- TimeStart: The TimeStart input is a DateTime object which specifies the start of the time range for which to backfill data. The TimeStart input is required when backfilling data.
- TimeEnd: The TimeEnd input is a DateTime object which specifies the end of the time range for which to backfill data. The TimeEnd input is required when backfilling data.

The QueueDataBackfill method will queue a backfill for the specified tag for samples with a time greater than or equal to TimeStart and less than TimeEnd. If TimeStart or TimeEnd has a DateTimeKind of Local or Unspecified the time will be converted to UTC. QueueDataBackfill will throw an exception if an attempt is made to queue a backfill for an invalid TagID.

5.4 Data Access Methodologies

To facilitate access to the client historian when data is requested by an end user a suitable data access methodology needs to be chosen. This section details two potential solutions compares their advantages and drawbacks.

5.4.1 Pass-through

The pass-through methodology is the simpler of the two potential solutions. In this solution there is no local storage component, with all data being accessed from the historian in real-time as requests for data are received. The steps required to obtain data when requested for a report are listed below and shown in Figure 12.

1. User requests a report to be run. The report generation code determines the data required to build the chosen report and requests this data from the middleware via the middleware common interface.
2. The middleware determines the appropriate method to retrieve the data from the client historian, then connects and makes a request to the historian for the required data.
3. When the required data has been collated by the client historian it is forwarded back to the middleware using the connection opened in the previous step.
4. The data from the historian is reformatted by the middleware as required to match the format used in the middleware common interface and returned to the report generation code for inclusion in the report.

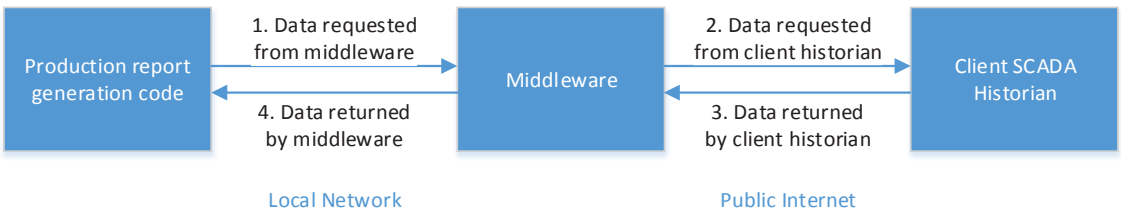

Figure 12: Pass-through connection data access

The pass-through methodology has the following advantages:

- The data obtained via the middleware is always up to date as it is retrieved in real-time from the historian. If a report was to be run multiple times in sequence each subsequent copy would contain more data than the previous report.

- The middleware does not need access to any significant storage as the data is not stored within the local network. This consequentially means a reduced hardware requirement.
- The middleware does not require any backfilling abilities as there is no local data store to backfill data into.

These advantages are countered by the following drawbacks:

- There is a high potential for latency when accessing data as the request needs to leave the local network and traverse the public internet, where in all likelihood the bandwidth available is much reduced when compared to the local network. The client historian could also be located on a connection at a treatment plant with limited bandwidth available.
- The pass-through methodology provides no redundancy for the data as the only copy resides on the client historian. In the event that the client historian is not running it would not be possible to access SCADA data for reports.
- Access to data for reporting is reliant on the client internet connection being online at the time a report is accessed. If the client connection is down it would not be possible to access SCADA data for reports.

The drawbacks of this solution make it an unsuitable implementation. The middleware needs to provide redundancy to protect against data loss and have the ability to provide data regardless of the state of the client internet connection.

5.4.2 Collection and Storage

The collection and storage methodology is the more complex implementation of the two potential solutions. In this solution the data collected from the historian is stored in a local database, with all data being retrieved locally during report generation. The operation of the solution is split into two distinct parts; the data collection and storage, and the stored data access. The steps required for data collection and storage are listed below and shown in Figure 13.

1. On a periodic basis the middleware requests data from the client historian for the time period that has just passed. The middleware determines the appropriate method for accessing the data, then connects and makes the request to the historian.
2. When the required data has been collated by the client historian it is forwarded back to the middleware using the connection opened in the previous step.

3. The data from the historian is reformatted by the middleware as required to match the format used in the middleware database and stored locally for access as required at a later time.

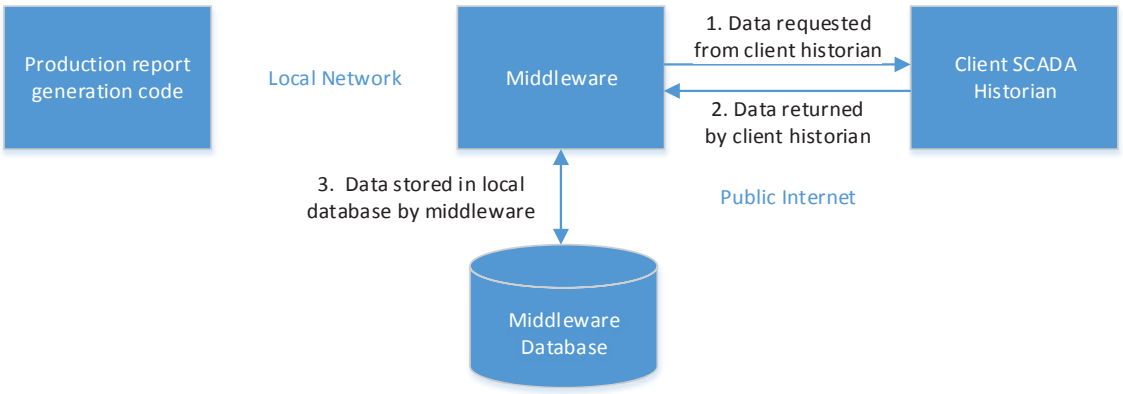

Figure 13: Data collection and storage

The second part of the solution consists of stored data access. The steps required for data access are listed below and shown in Figure 14.

1. User requests a report to be run. The report generation code determines the data required to build the chosen report and requests this data from the middleware via the middleware common interface.
2. The middleware retrieves the requested data from the middleware database using the local network.
3. The data from the database is reformatted by the middleware as required to match the format used in the middleware common interface and returned to the report generation code for inclusion in the report.

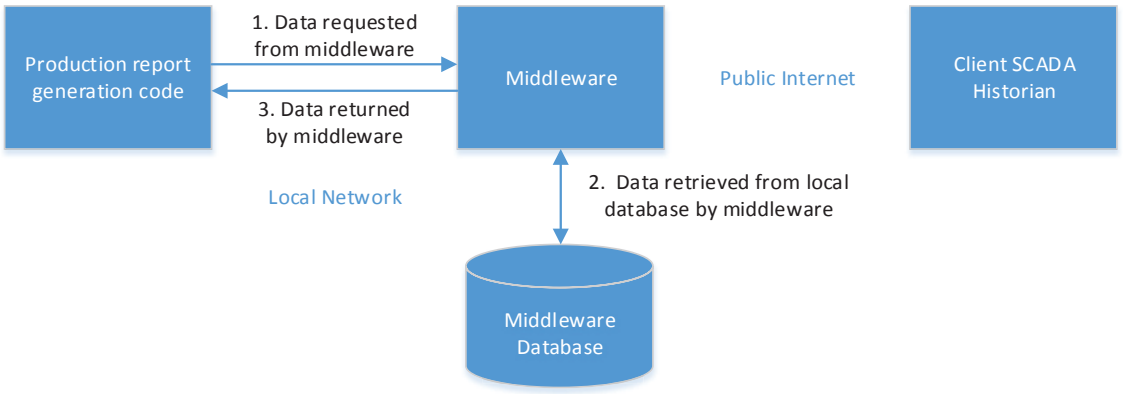

Figure 14: Stored data access

The collection and storage methodology has the following advantages:

- The data required for the report is returned quickly as the data is all accessed via the internal network. There is no communication via the public internet or client network infrastructure which could significantly slow data access.
- The middleware can return data from the middleware database regardless of the status of the internet connection. In the event the client internet connection is down the middleware database would contain data until shortly before the connection was lost.
- The local database contains a mirror of data that is required by the client for compliance reporting. In the event the client historian fails the compliance data is mirrored and still accessible.
- The client no longer needs to maintain legacy historians for the purposes of reporting. Should the client upgrade to a newer or different SCADA solution the old historian can be removed as the compliance data it contained is stored in the middleware database.

These advantages are countered by the following drawbacks:

- A large amount of local storage may be required for the middleware database to hold the required compliance data.
- As the middleware retrieves data in blocks at a specific time interval from the historian the most up to date data may not be in the middleware database when the report is requested. A report run just before the middleware retrieves data for a just-passed time interval will not contain any data for that time interval.
- The middleware needs the ability to backfill historic data from the historian to the local database. Backfilling would be required to retrieve historic data that existed in the historian before implementation of the middleware, and to populate data that may not have been present in the historian for a given time interval when that interval was requested by the middleware.

Despite these disadvantages the collection and storage methodology meets more of the implementation requirements than the pass-through methodology. Therefore in the implementation of the middleware the collection and storage model will be used.

5.5 Reliable Data Collection

The reliable collection of data is a very important consideration for this project. The data collected is used to ensure compliance with water standards and therefore needs to be available

after recovery from interruptions such as losses of connectivity. This section explores potential options that could help make data collection more reliable.

5.5.1 Data Integrity Checking

To ensure all data has been collected from the client historian an integrity check could be implemented. The integrity check would take place at the end of a given block of time, querying the historian to ensure the local database contains all of the information for the tag that the historian database does. This would entail the middleware querying all data for the block of time from the local database and comparing this sample by sample to the data returned by the client historian. Any missing samples would be added to the local database as they are detected to be missing. Figure 15 shows an implementation of a simple integrity checking scheme.

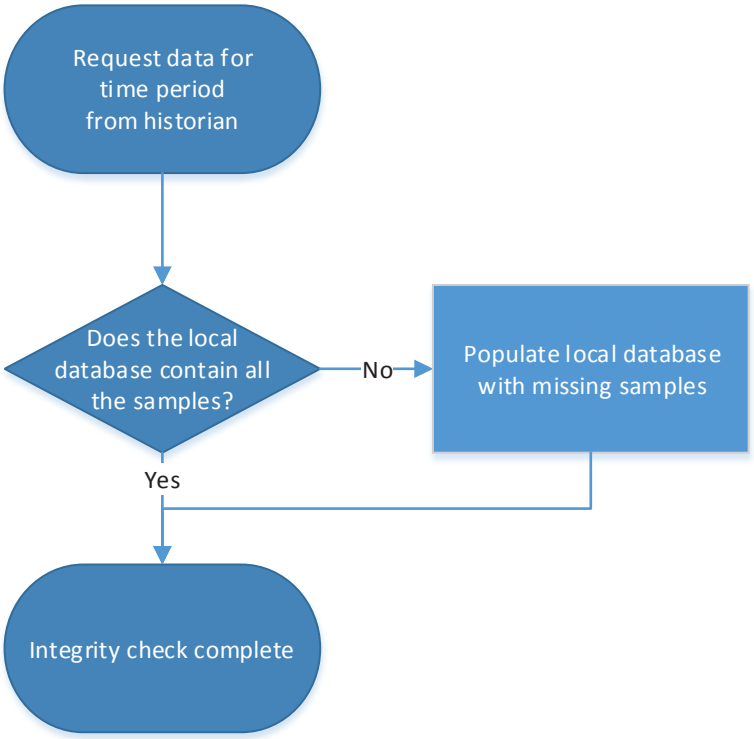

Figure 15: Simple data integrity checking scheme

Ideally this check would take place a reasonable duration after the conclusion of a given time block to give data that may have been delayed and missed initial collection to arrive at the client historian.

While this simple scheme meets the requirement of maintaining data collection reliability it necessitates all data be requested from the historian twice, causing unnecessary data transfer. The implantation would also have a high resource overhead as the middleware would be

required to iterate over all of data collected for the time block to determine which data was missed during the initial collection.

To improve the speed and reduce bandwidth requirements the algorithm could be modified to compare the count of samples for the time block in the local database with the count of samples in the historian database for the same time block. The ability to query the sample count from the historian varies between implementations. In the case that the historian doesn't support giving the sample count for a time block the middleware would need to request all of the data for that time period from the historian and perform the count internally.

5.5.2 Last Connection Time Tracking

When collecting data from the client historian sequentially a reliable interface could be achieved by tracking on a per-tag basis the last time at which data was successfully downloaded. This method requires an additional data field to be stored per tag row in the middleware database. When data is successfully downloaded for a given time block from the historian the last connection time field is updated with the time corresponding to the end of the successfully downloaded time block. When collection for the next time block is required the date range used for collection would start at the last successful collection time and end with the requested time block. This approach would ensure that any data missed in previous data collection operations would have an attempt made for collection with every new time block until successfully downloaded. A simple implementation of the last connection time tracking scheme is shown in Figure 16.

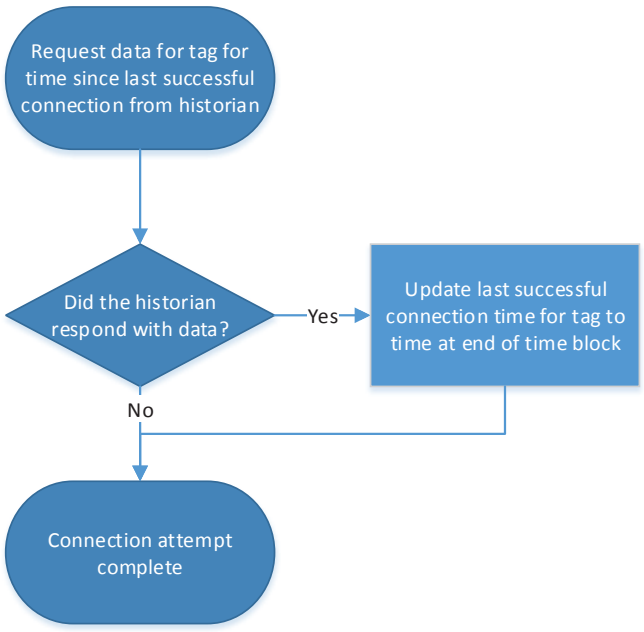

Figure 16: Last connection time tracking scheme

There are some drawbacks to the last connection time tracking scheme. To ensure timely collection of data for reporting each time block would need to be collected reasonably soon after it has elapsed. While this in itself poses no problems issues could arise if the source data is slightly delayed arriving at the historian. The data may still be in transport to the historian when the middleware performs collection. As the collection would succeed and the last connection time would be updated the missing data would not be retrieved.

The last connection time tracking implementation does not provide an inbuilt mechanism for backfilling of historic or missed data. A separate system would need to be implemented to facilitate the backfilling of historic or missed data from the historian. These limitations result in a system that is not suitable for implementation in the middleware.

5.5.3 Queued Data Collection

Storing transactions in a persistent queue provides a reliable means of data collection. Sometime shortly after a given time block has elapsed a data sync transaction for each monitored tag is enqueued into a first-in first-out queue. The queue item contains details about which tag the transaction is for, and the start and end time of the time block. An example queue is shown in Figure 17.

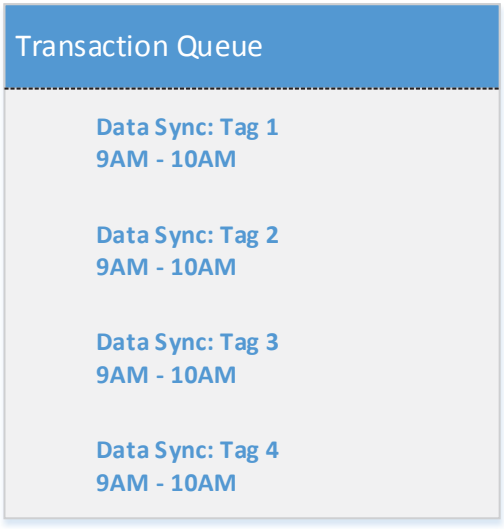

Figure 17: Example queue showing data sync operations

A queue item processor is implemented to process transactions from the queue. The item processor removes the first item from the output end of the queue and attempts to perform the operation it specifies. If the queue item fails it is enqueued at the end of the queue to be attempted next time the queue is processed. The operation of the queue processor is shown in Figure 18.

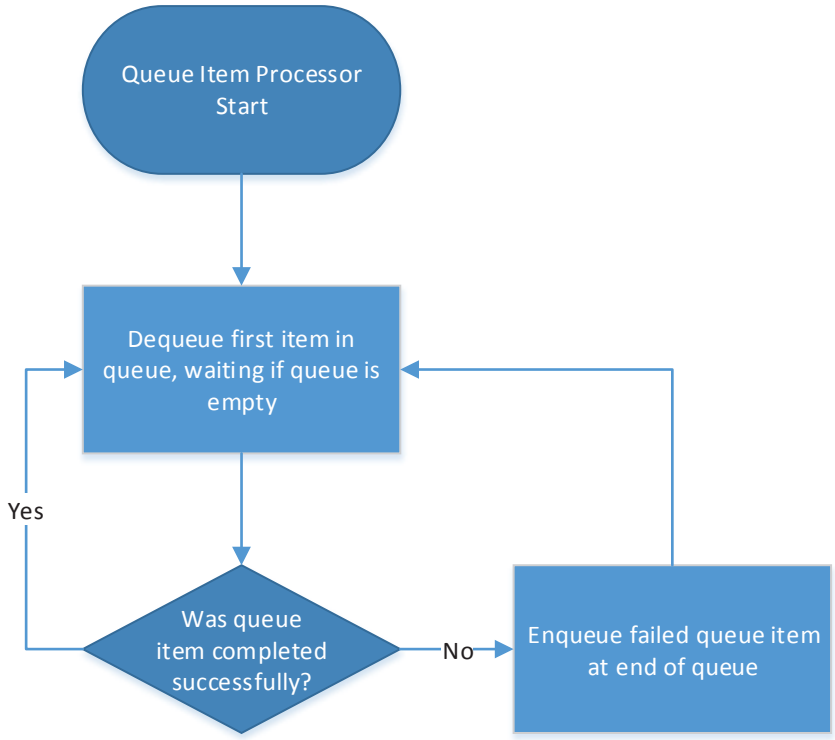

Figure 18: Queue item processing

This implementation provides a reliable solution for backfilling data. Queue items for backfills can be inserted as queue items with historic times when required. The backfill items will be processed by the queue data processor as would a queue item for current data collection and would be continually enqueued until the backfill has succeeded.

The queue still does not solve the issue of potential data loss occurring if the data sync queue items are queued soon after a time block has elapsed. As with the last connection time tracking solution data that arrives at the client historian late may not be collected.

To ensure maximum reliability any implementation of the queue system would need to ensure that if the application was terminated or stopped unexpectedly the in-progress queue item would remain in the queue and not be lost.

5.5.4 Queued Data Collection and Integrity Checking

Both the data integrity checking and queued data collection approaches have shortcomings that make them unsuitable for implementation in the middleware. However combing aspects of both solutions provides an option that meets the requirements for use in the middleware.

The queued data sync is preserved from queued data collection option, with data sync transactions still being placed in the queue soon after a given time block has elapsed. To improve reliability an additional type of queue item for checking data integrity is added. The

data integrity check is as described in the data integrity check methodology with the count of samples available in the historian compared to the count of samples in the local database. As opposed to the data sync which is queued with only a small delay to real-time the data integrity check is queued for a time block of a few hours prior. An example of a queue containing both data sync and data integrity check transactions is shown in Figure 19.

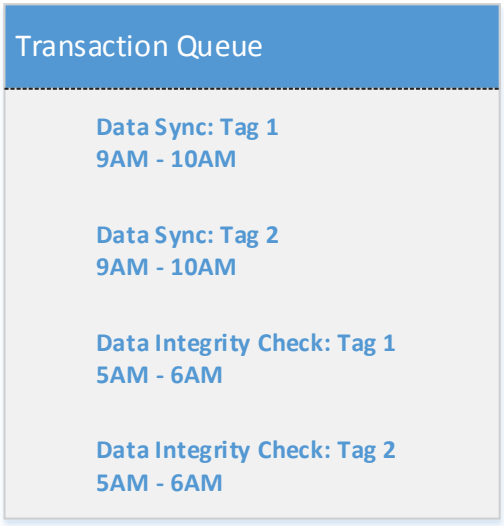

Figure 19: Example queue showing data sync and data integrity check operations

As the integrity check lags the data sync by a number of hours any data that has been delayed in transit or reported late due to clock skew should have subsequently arrived at the historian. If the integrity check fails a new data sync transaction for the failing time block will be queued to allow the missing data to be retrieved. This provides both timely data collection and protection against delayed data.

This solution still provides the same ability to queue backfill operations as in the queued data collection implementation. The queued data collection with queued data integrity checking will be the solution implemented during middleware development.

5.6 Secure Data Collection

As data collection from client historians involves accessing servers residing on local council networks security is paramount. The methodology used to secure access to the historian and the data while traversing the public internet will vary between councils depending on requirements and capabilities of their IT staff and infrastructure. Accordingly this section presents a number of potential approaches that will be used on a case by case basis as the middleware is deployed in the production environment.

5.6.1 Source Address Whitelisting

A simple yet reasonably secure methodology is for the client to filter connections as they arrive at the client firewall. This requires the client to maintain a whitelist containing only the source IP addresses of machines in the production environment. When requests arrive from production machines the request will be forwarded to the historian, while machines not in the whitelist will have their requests dropped by the firewall before entering the client network. An example of incoming connections from both a whitelisted and non-whitelisted computer are shown in Figure 20.

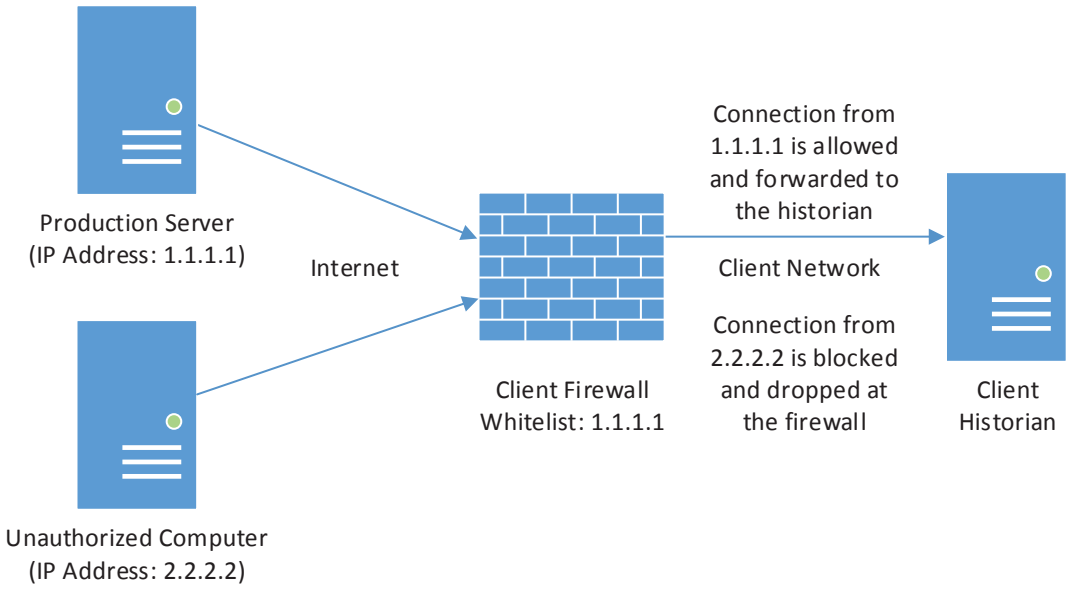

Figure 20: Source address whitelisted connection

The whitelisting methodology prevents connections originating from non-authorized sources and should therefore prevent connections being made to the historian from other internet users. However the whitelisting provides no other form of authentication, meaning any requests from the whitelisted addresses will be trusted. Also the data requested by authorized sources is not encrypted, and therefore could potentially be intercepted while en-route between the client historian and production server. While whitelisting may be acceptable as the sole means of security to some clients it would be best used in conjunction with another form of security to provide additional protections.

5.6.2 Authentication

Requiring credentials or another form of pre-shared key provides a means of restricting connections. Any requests made from a given source would first be required to authenticate before any data would be returned. The authentication scheme generally used by SCADA

historians is a username and password combination which would be configured on the historian and passed back to be populated in the middleware.

Utilizing just authentication does provide an adequate level of security but does have some potential problems. If the connection to the historian is not encrypted it is possible that the credentials and data could be captured while en-route to the historian, negating all security as the credentials could then be used by unauthorized users.

If connections are allowed from any internet user it is also possible a brute-force attack could be used to obtain a valid set of credentials. To brute-force the credentials an attacker would start an automated process which would continually attempt to connect with new credentials until a working set is found. These attacks are often successful where common words are used in a password, or where the password is insufficiently long.

5.6.3 Source Address Whitelisting with Authentication

Combining source address whitelisting with authentication provides a higher level of security than if either scheme was used alone. As attempts to authenticate would only be allowed from a whitelisted set of trusted IP addresses the risk of a brute-force attack is mitigated as an external attacker would not be able to connect to the historian to attempt authentication. The credentials and data could still be intercepted while en-route over the public internet which could be a security risk if the login credentials are used elsewhere.

5.6.4 SSL Encryption

To prevent data and credentials being intercepted while traversing the public internet it is possible in some cases to use Secure Sockets Layer (SSL) to encrypt the connection. As Microsoft SQL Server supports SSL connections from clients any of the numerous historians utilizing SQL Server as a data store could have SSL enabled. SSL is also used as part of the secure HTTPS standard, potentially allowing historians utilizing a HTTP based protocol to be secured.

Ideally SSL would be used in conjunction with both source address whitelisting and authentication. The SSL connection would prevent the authentication credentials and data being intercepted while en-route, and the source address whitelisting negates brute-force attacks on credentials. SSL could be used with only a single complementary security method though the implementation would not be entirely secure.

5.6.5 VPN Tunnelling

In some cases the client council may be unwilling to open their firewall to incoming connections from the internet. To facilitate connections to historians it is possible the client could initiate an

outgoing virtual private network (VPN) connection back to the production environment over which data is accessed. If the client already has VPN infrastructure in place it would also be possible for a VPN connection to be initiated from the production environment to the client.

In a typical VPN implementation the connecting party is provided with a pre-shared key which could take the form of authentication credentials or certificate. The connection could be initiated from a hardware device such as a router with inbuilt VPN support, or from the historian or other server via an existing router. The VPN connection would then terminate on either a hardware device supporting VPN connections or a server providing a VPN endpoint. Once the VPN connection has been established traffic could travel in either direction between networks, allowing the middleware to make requests of the historian regardless of where the VPN connection originated. A typical VPN implementation is shown in Figure 21.

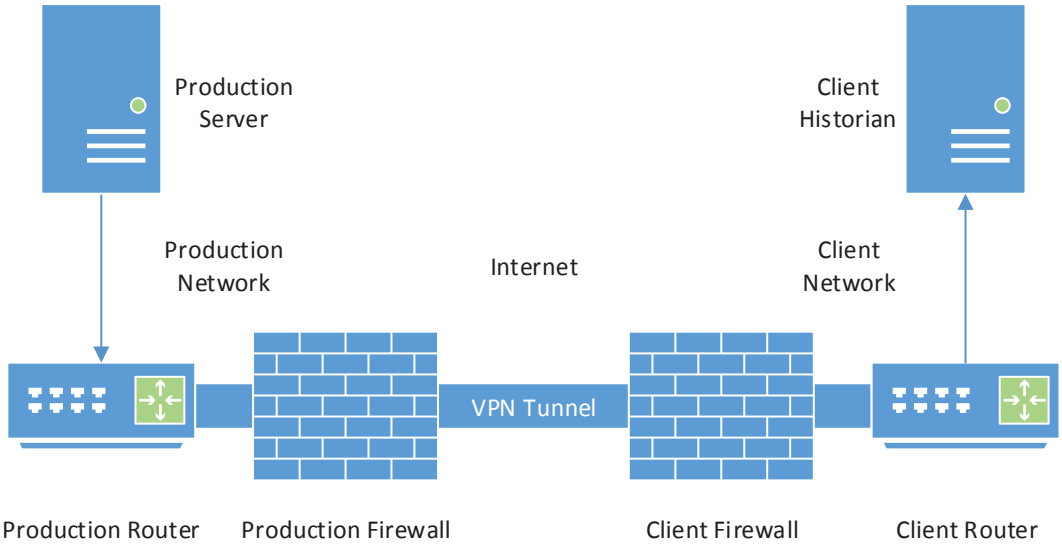

Figure 21: VPN tunnelled connection

A number of VPN technologies exist and the appropriate implementation would be selected in conjunction with the client. These range from software only solutions to hardware with inbuilt VPN clients and servers. No-cost options are available with the open-source OpenVPN software providing an excellent feature set and compatibility across many platforms.

As the VPN tunnel created between the production environment and the client network is encrypted the security of data traversing the public internet is maintained. Additional security can also be achieved by limiting the source IP addresses allowed to connect as in the source address whitelisting methodology. Authentication and encryption are part of the VPN implementation, so when combined with appropriate firewall whitelisting using a VPN provides an ideal solution encompassing a number of security methodologies.

5.7 Data Compression

As part of this research a large dataset containing approximately ten years of historic data was obtained from a New Zealand council. This was in the form of an SQL database containing data collected from their historians via a tool they had developed in-house. The historians used by the council had been collecting data from both water and wastewater treatment plants. The samples in the database were raw data as read from the SCADA instrumentation, generally polled at a fixed interval. Experiments were conducted with this data to determine the level of compression provided by two simple compression schemes. The compression schemes were chosen as they can be utilized with the SQL Server databases as used in the production environment. The results of these tests are documented in this section.

5.7.1 Sampling Deadband

The first form of compression tested consisted of varying deadbands being applied to the sample data. A deadband prevents a new sample being logged for a given tag until the current value has changed by a specified amount or more. An example of samples taken with a 10% deadband is shown in Figure 22.

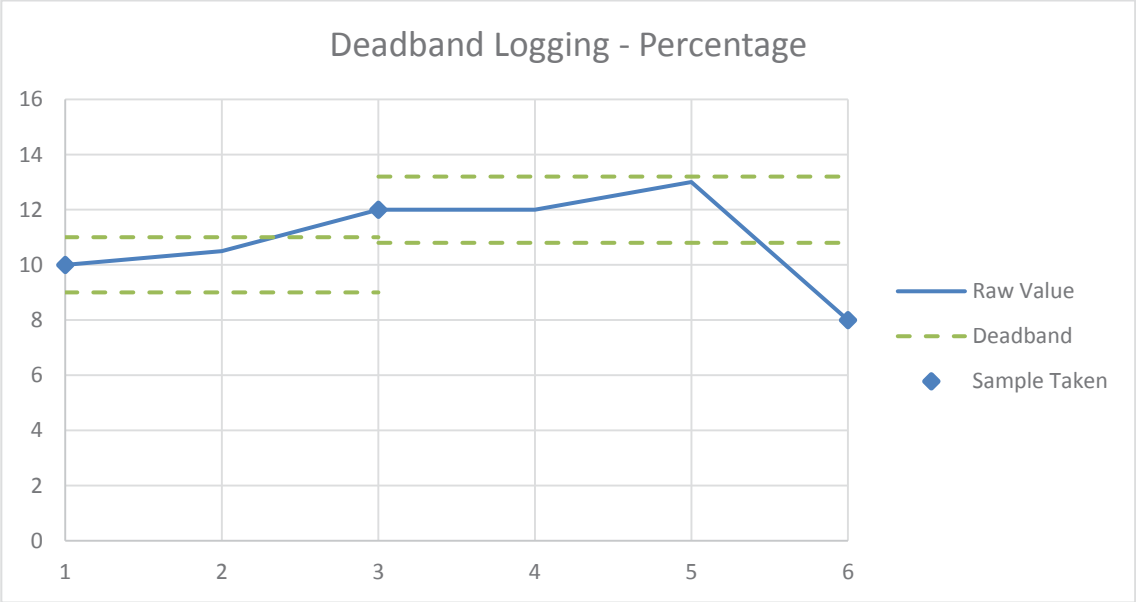

Figure 22: Sampling with a 10% deadband

In this example a new sample is not logged until the value has changed by at least 10% when compared to the last logged value. The changes in raw value are shown by the solid line, the current deadband by the dotted line, and logged samples by the diamond marker.

A deadband can be specified as either a percentage or fixed value. As an example the data shown in Figure 22 is shown resampled with a deadband of 1 in Figure 23. The small change in deadband results in an additional sample being logged.

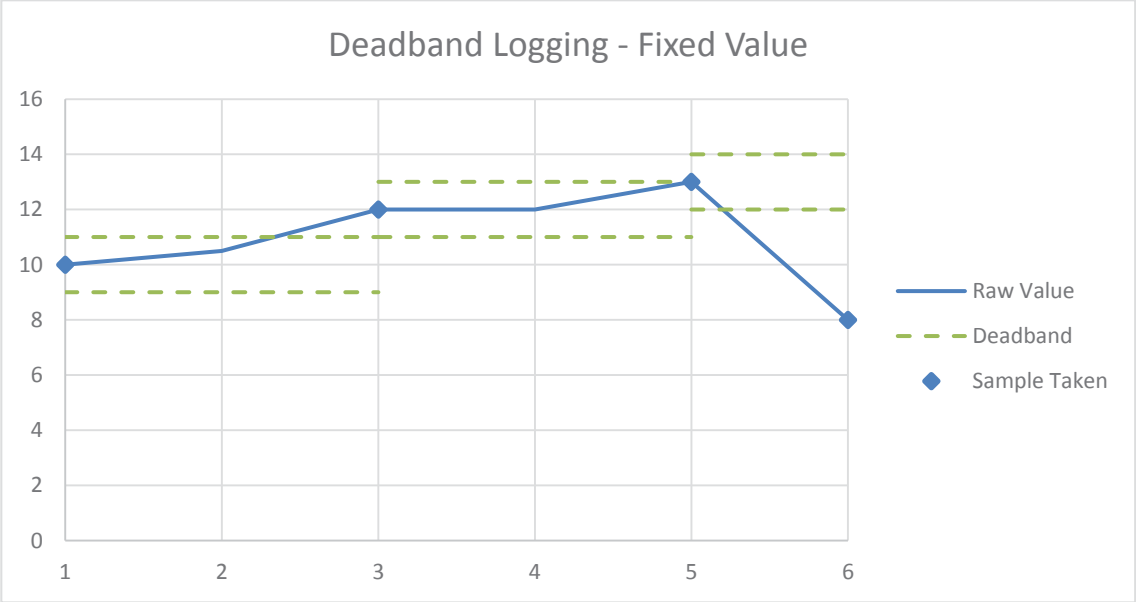

Figure 23: Sampling with a fixed deadband of 1

To test the compression achieved with varying deadbands the large dataset was compressed with four different deadbands of 0.5%, 1%, 5% and 10%. The dataset contained approximately 1.8 billion samples collected from approximately 670 tags. The data was sourced from both water and wastewater treatment facilities, and represented approximately 12 years of data. The number of samples left after compression is shown in Figure 24. As can be seen the number of samples required after compression decreases dramatically as the size of the deadband increases.

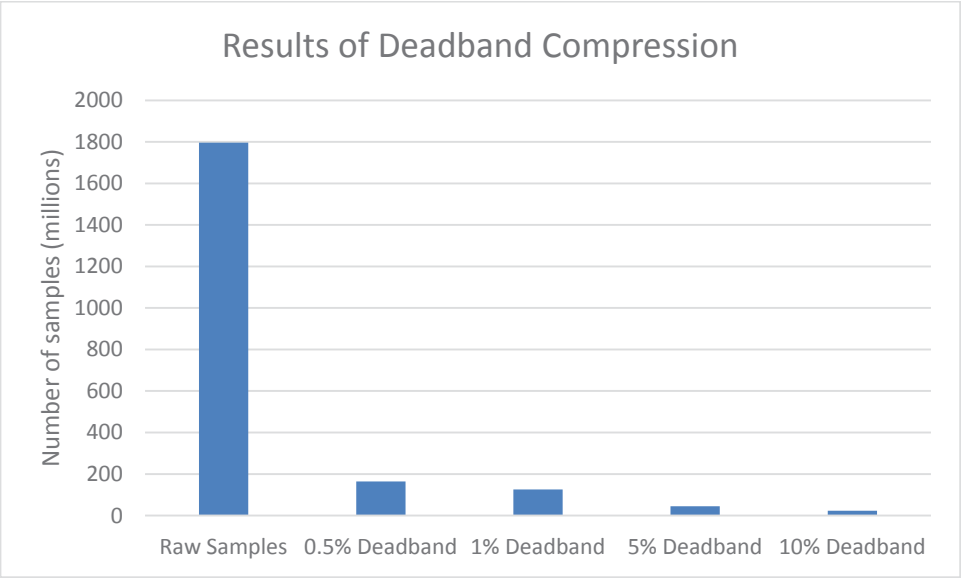

Figure 24: Results of compression with varying deadbands

While the deadband compression produces excellent results the precision of the data is altered. As the data in the production environment is used for compliance reasons it cannot be modified arbitrarily – any decisions about reductions in accuracy of the data would need to be made by the client. While it would be possible to allow the client to set the deadband on a tag by tag basis this would be a time consuming process and would therefore be potentially left unconfigured. Allowing the deadband to be configured could also lead to undesired data loss in the case that a deadband was inadvertently configured too high. For these reasons it is unlikely utilizing deadband compression will work reliably in the middleware when used in the production environment.

5.7.2 Sample Deduplication

Another potential approach for compression is to have the middleware remove repeated samples. Removing the repeats will have no effect on the quality of data as during analysis the production report generation code carries the last known sample forward until the next sample. An example of the report generation algorithm with raw data is shown in Figure 25, with the same data after deduplication shown in Figure 26.

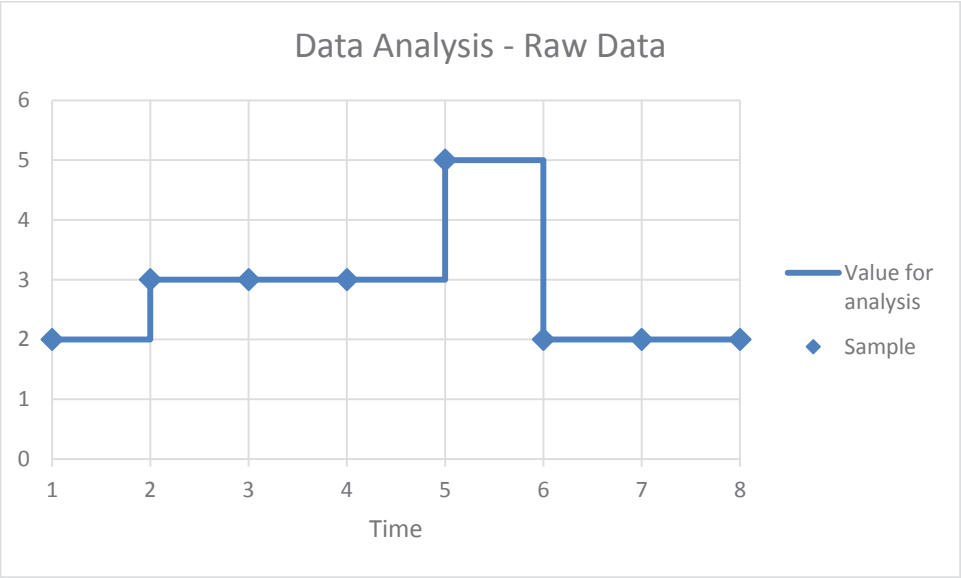

Figure 25: Report generation with raw data

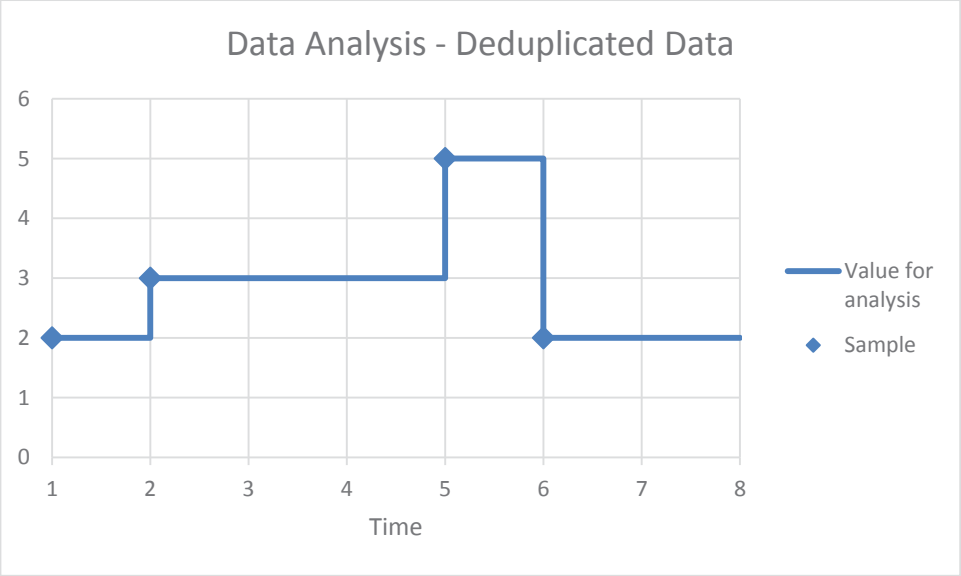

Figure 26: Report generation with deduplicated data

As can be seen the value used for data analysis during report generation is not changed by the removal of duplicate samples. In this example the number of samples is reduced from eight to four with no change in the output data.

To test the compression attained on real-world data using deduplication the sample dataset of approximately 1.8 billion samples was processed. While the compression provided was significantly less than using a sampling deadband the data compression was lossless yet still provided a dataset less than a third of the original size, being 29% the size of the original. The compression deduplication achieved is shown in Figure 27.

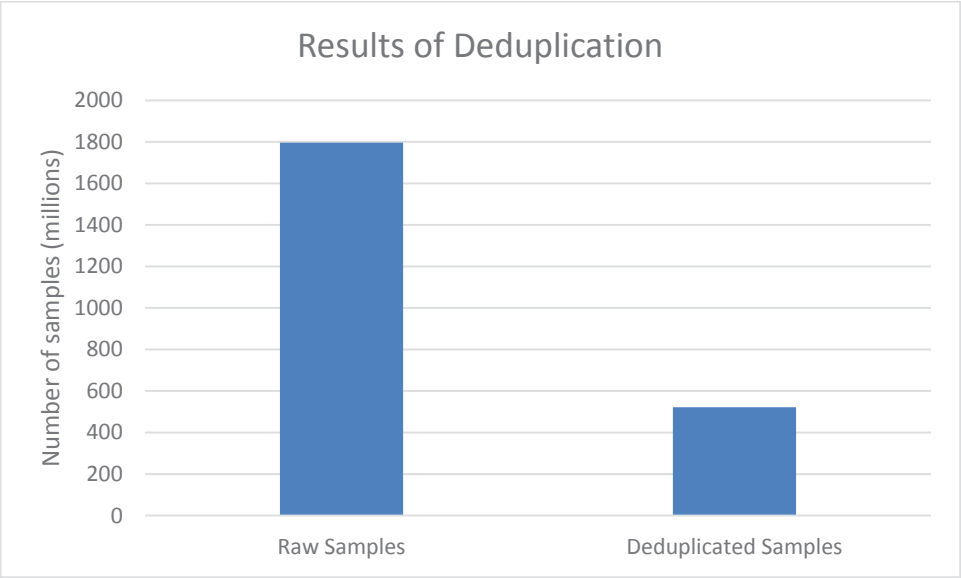

Figure 27: Results of compression with sample deduplication

While the compression ratio achieved using a sampling deadband would be desirable in the middleware implementation the loss of precision would result in data that may no longer be suitable for compliance purposes. While the compression with deduplication is reduced it still provides a significant reduction in storage requirements without any loss of precision. For this reason deduplication provides an ideal solution for implementation in the middleware.

Chapter 6: Middleware Implementation

This section details the implementation of the middleware using the methodologies determined to be appropriate in the previous section.

6.1 Application Runtime Environment

To ensure maximum compatibility with the production environment C# 4.0 from the .NET Framework 4.0 was selected for application development. The .NET Framework 4.0 is already part of the production virtual machine image and already has a methodology for security patches and updates in place. The production web interface and reporting code have also been developed using the .NET Framework 4.0, allowing simple integration of the common interface assembly which will facilitate access to the middleware. Microsoft Visual Studio 2010 was selected for use in development as it integrates tightly with the .NET Framework 4.0.

Microsoft SQL Server 2008 R2 was selected as the database for the middleware implementation. SQL Server 2008 R2 is available as part of the production virtual machine image and is used for the production web interface and reporting system, meaning it is available for all existing client installations. A method for security patches and updates is already in place, therefore requiring no new software or procedures be put in place for patch management. The .NET Framework 4.0 provides an inbuilt data provider for SQL Server allowing direct access to databases with no additional software.

6.2 Middleware Architecture

The middleware is implemented in three distinct components - the Windows service, the middleware database and the common interface assembly. Figure 28 shows the relationship between these components, the SCADA historians and client applications which access the middleware data.

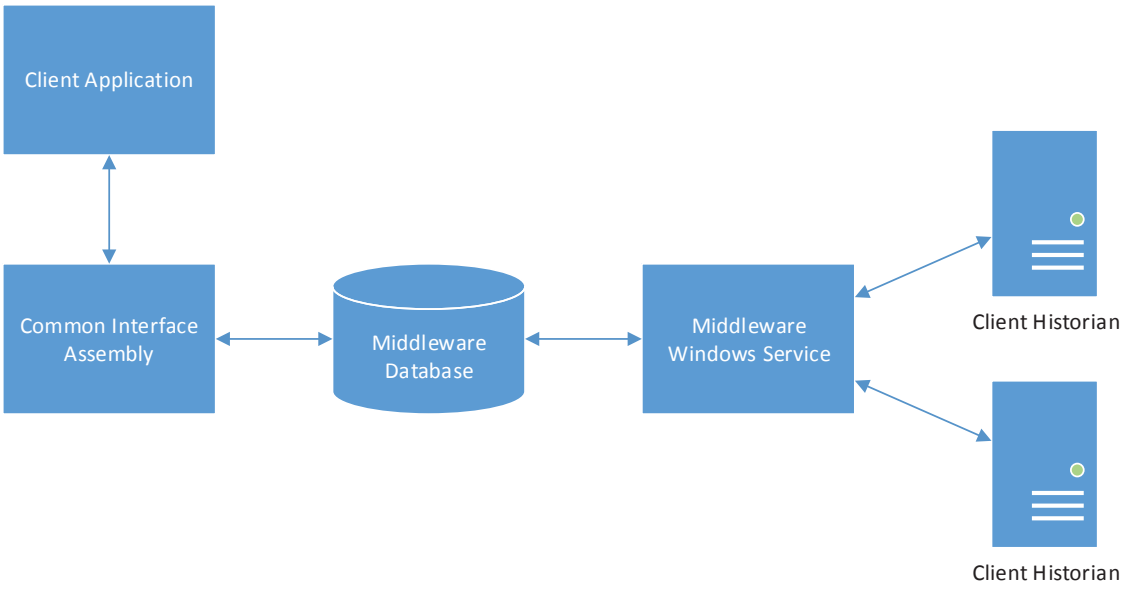

Figure 28: Implemented middleware architecture

6.2.1 Middleware Windows Service

The data collection component of the middleware is implemented as a Windows service. The Windows service was chosen as services have the ability to run unattended and headless, allowing the middleware service to run on start-up of the virtual machine operating system even when no user is logged in. This unattended start-up is required as the virtual machine operating system will be restarted by an automated process periodically for the installation of updates. After this restart the middleware service needs to start without user intervention. Running as a service provides additional resiliency as the service can be automatically restarted by Windows in the event that it unexpectedly stops running.

For testing purposes a traditional application version of the middleware was also implemented. In the event that the service executable is started by a user in an interactive desktop session the middleware will run as an application until closed by the user. This allows testing to be performed in an interactive environment where trace output from the middleware application can be seen in a console window.

6.2.2 Middleware Database

The middleware data is stored in an SQL Server database. This database can reside on either the local machine or a remote machine as all communication between the middleware and the database are performed over a TCP/IP network connection, though in the production environment the middleware database will generally reside on the virtual machine that is running the middleware service. More information regarding the database schema is provided in the next section.

6.2.3 Common Interface Assembly

To facilitate client applications accessing data that has been collected in the SQL database by the middleware service the common interface assembly is used. The assembly is written in C# and can therefore be included in applications written using the .NET Framework. The assembly provides methods as listed in section 5.3 to give access to the data.

The assembly communicates directly with the SQL Server database to retrieve data and queue requests for data backfilling. This removes the need for a custom data access protocol to be provided in the middleware service. As the middleware service isn't required to accept incoming network connections it does not need firewall rules to be added to the host running the service.

6.3 Database Schema

The middleware database contains five tables which provide the required data storage for the middleware. This includes storage for information about client historians, client historian tags and samples collected from these historians. Database tables are also used for tracking pending queue transactions and configuration information for the middleware. These tables are shown in Figure 29 and explained in detail in this section.

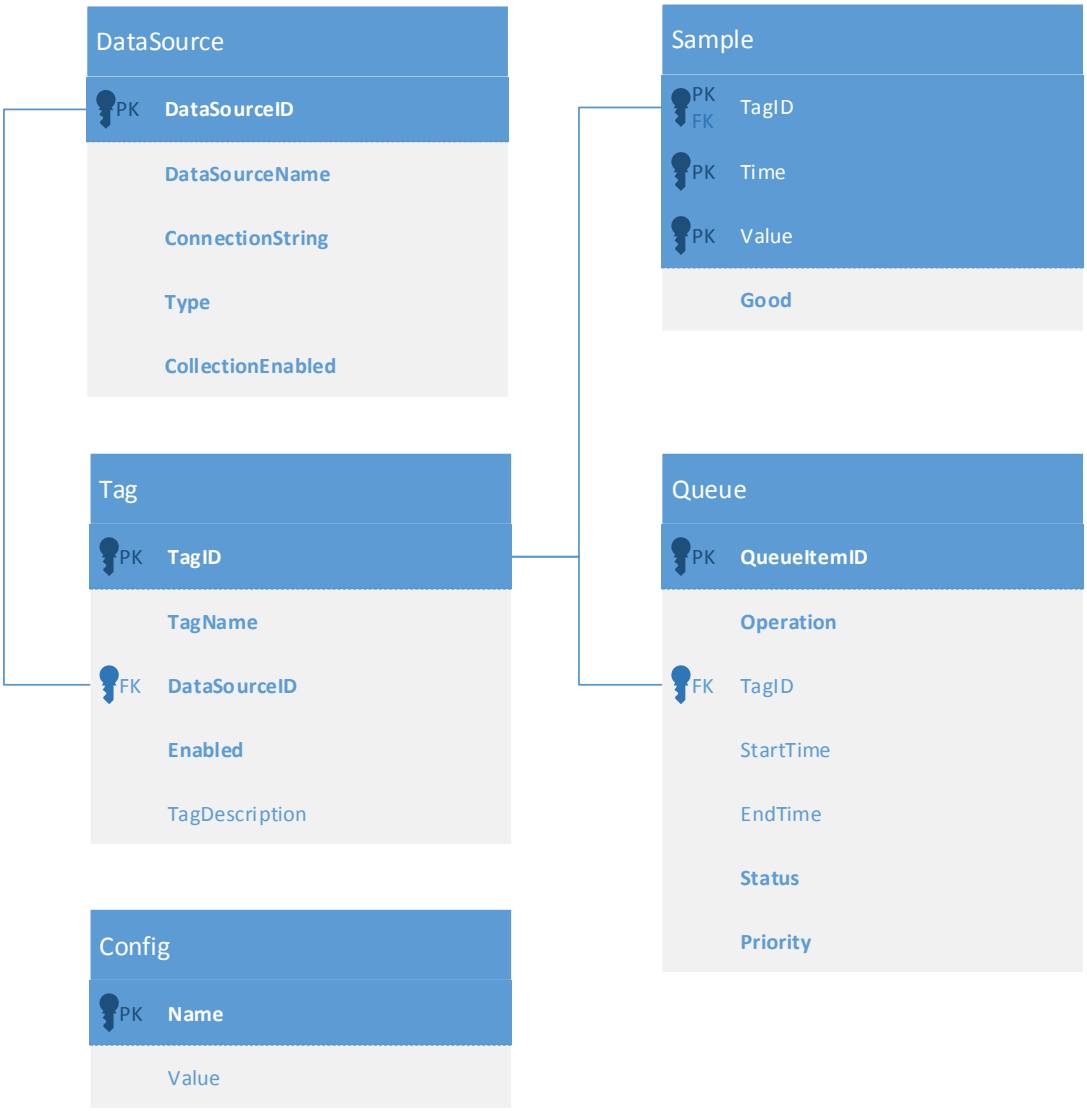

Figure 29: Middleware database schema

6.3.1 DataSource Table

The DataSource table contains information about the client historians to which the middleware will connect and retrieve data from. The columns in this table are detailed below.

- DataSourceID: This column is an integer (PK, int, not null) which forms the primary key for the table. The value for this column is automatically generated when inserting a new row in the table. The DataSourceID column will always contain a value.
- DataSourceName: This column is a string (nvarchar(max), not null) which represents the name of the data source. The value for this column is entered by the operator when configuring a new data source. The DataSourceName column will always contain a value.

- ConnectionString: This column is a string (nvarchar(max), not null) which contains the connection string for the client historian. The connection string generally contains a semicolon delimited collection of key/value pairs which contain information about the client historian needed by the middleware when connecting. The ConnectionString column will always contain a value.
- Type: This column is a string (nvarchar(50), not null) which contains a string specifying the type of client historian to which the row refers. The middleware uses this value to select the appropriate data provider when initiating a connection. The type column will always contain a value.
- CollectionEnabled: This column is a Boolean (bit, not null) which specifies whether the middleware should attempt to collect data from the client historian. When the value is 1 the middleware will queue data collection operations for the historian. When the value is 0 the middleware will not queue data collection operations. This will typically be set to 0 when a historian is decommissioned but it is required the previously collected data remains in the middleware database.

Rows in the DataSource table will generally be retrieved via the DataSourceID key, or enumerated as an entire table. Therefore the index for the table consists only of the DataSourceID column.

6.3.2 Tag Table

The tag table contains a list of all tags currently available from the client historians in the DataSource table, along with tag status information. The columns in this table are detailed below.

- TagID: This column is an integer (PK, int, not null) which forms the primary key for the table. The value for this column is automatically generated when inserting a new row in the table. The TagID column will always contain a value.
- TagName: This column is a string (nvarchar(max), not null) which represents the name of the SCADA tag. Depending on the type of historian from which the tag originates this column may contain the tag name or the tag name and location. The TagName column will always contain a value.
- DataSourceID: This column is an integer (FK, int, not null) which provides a foreign key to the data source row for the corresponding client historian. The DataSourceID column will always contain a value.

- Enabled: This column is a Boolean (bit, not null) which specifies whether the middleware should attempt to collect data for the tag in the case that the corresponding client historian data source is enabled. Enabled can be set to 0 resulting in no data collection for tags that are not required for compliance or tags that are decommissioned and are no longer generating data. Setting Enabled to 1 will allow data collection for the tag.
- TagDescription: This column is a string (nvarchar(max), null) which contains a description of the tag where available from the client historian. Depending on the type of client historian this field may contain a description as entered by the client, or other information such as units of measurement which may be reported by the historian. In some cases this column may contain a null value where no extra information is available from the historian.

Rows in the Tag table will be added by the middleware as new tags are discovered on client historians. References to rows in the Tag table will generally be made using the TagID column from other tables so the TagID column is used as the sole index for the table.

6.3.3 Sample Table

The sample table contains all samples retrieved from client historians and represents the majority of data stored in the middleware database. The columns in this table are detailed below.

- TagID: This column is an integer (PK, FK, int, not null) which forms part of the primary key for the table. This column is a foreign key associating the sample value with the corresponding tag in the Tag table. The TagID column will always contain a value.
- Time: This column is a DateTime (PK, datetime2(7), not null) which contains the time at which a given sample was generated. The column forms part of the primary key for the table. The sample time is stored in coordinated universal time (UTC) to ensure a continuous samples without gaps or duplicate samples for time periods near daylight saving transitions. The Time column will always contain a value.
- Value: This column is a double precision float (PK, float, not null) that contains the sample value. The Value column forms part of the primary key for the table. The value column will always contain a value.
- Good: This column is a Boolean (bit, not null) which specifies whether the value of the sample is valid. When the value of the Good column is 0 the sample was reported as invalid by the client historian and should not be used for compliance purposes. Samples

with a value of 1 in the Good column are valid samples. The Good column will always contain a value.

Rows in the sample table will be populated by the middleware as they are collected from the client historian. To prevent the same sample being inserted twice the table uses a primary key consisting of the TagID, Time and Value columns. Any attempts to insert a sample for a given tag with an identical time and value to that of an existing sample will result in a primary key violation, preventing the sample being inserted in the middleware database more than once.

When data is retrieved from the database it will be queried via the common interface for a given tag over a specified time period. The Sample table therefore utilizes an index containing the TagID, Time and Value columns to ensure data can be located first by TagID, then time span.

6.3.4 Queue Table

The queue table contains a list of all operations queued for execution by the middleware. This includes retrieving data and a number of other operations detailed in section 6.5. The columns in this table are detailed below.

- QueueItemID: This column is an integer (PK, int, not null) which forms the primary key for the table. The value for this column is automatically generated when inserting a new row in the table. The QueueItemID column will always contain a value.
- Operation: This column is a string (nvarchar(50), not null) which specifies the type of queued operation to which the row corresponds. These operation types are detailed in section 6.5.1. The Operation column will always contain a value.
- TagID: This column is an integer (int, null) which may provide a foreign key to a tag row if the queued operation is specific to a tag. Some operations are not specific to a tag in which case the value of the TagID column will be null.
- StartTime: This column is a DateTime (datetime2(7), null) which may provide the start of a date range for the queued operation. Some operations do not require a time range in which case the value of the StartTime column will be null.
- EndTime: This column is a DateTime (datetime2(7), null) which may provide the end of a date range for the queued operation. Some operations do not require a time range in which case the value of the EndTime column will be null.
- Status: This column is an integer (int, not null) which corresponds to the current status of the queued operation. The valid status values are detailed in section 6.5.2. The Status column will always contain a value.

- Priority: This column is an integer (int, not null) which corresponds to the priority of the queued operation. The priorities used by the middleware for queue operations are detailed in section 6.5.3. The Priority column will always contain a value.

6.3.5 Config Table

The config table provides storage for configuration information used by the middleware. This includes data such as state information for data collection operations. Particular usage of configuration parameters are noted throughout this implementation section as required. The columns in this table are detailed below.

- Name: This column is a string (PK, nvarchar(50), not null) which contains a unique name for the configuration parameter. The Name column will always contain a value.
- Value: This column is a string (nvarchar(MAX), null) which contains a string representation of the value of the configuration parameter. The format of this string is detailed in section 6.4.1. The Value column may be null if a given configuration parameter does not currently have a value.

The configuration parameter name must be unique and is used as a primary key for the table to ensure this is always the case. Configuration parameters are always accessed using the parameter name.

6.4 Middleware Configuration

To ensure the middleware configuration is readily accessible it is stored in the same database as the collected samples in a table as described in section 6.3.5. This allows the configuration to be retained even when the middleware is running on a different machine to the database server. Storing the configuration in the database also allows the existing database management tools to be used to update the middleware configuration when required.

6.4.1 Configuration Data Types

The configuration database allows a number of different data types to be stored. For some simple data types the configuration value is stored as a plain text string, with more complex configuration values being stored as a serialized XML string. This section details the storage of these configuration value data types.

- String: As the configuration database table uses a string to store all configuration values strings are simply inserted in the configuration table without change.

- Boolean: Boolean values are stored using the string 'True' to represent an enabled state and the string 'False' to represent a disabled state.
- Integer: Integers are stored using the string representation of the integer value.
- Double: Doubles are stored using the string representation of the double value.
- DateTime: Date/time values are stored using a UTC combined date and time format with local time zone offset as described in ISO 8601.

All other .NET objects that may need to be persisted in the configuration table by the middleware are stored as XML as generated by the .NET XML serializer.

6.4.2 Configuration Parameters

The set of configuration parameters essential to the operation of the middleware are described in this section.

- CheckWaitMinutes: This configuration parameter is an integer that sets the number of minutes the middleware will wait following the end of a given day before queuing data integrity checks for the data collected that day. The default value for this parameter is 10 minutes.
- ChunkMinutes: This configuration parameter is an integer that sets the size of the time block in minutes that the middleware will request for a given tag from the client historian when performing automatic data synchronisation. The default value for this parameter is 30 minutes.
- DatabaseVersion: This configuration parameter is an integer that stores the current schema version of the database. When a database update is required by new middleware code the database schema performs the update on start-up then updates the DatabaseVersion value to match the current schema.
- InstanceName: This configuration parameter is a string that contains a name for the running instance of the middleware. This is reported to the status monitoring code to allow the source of error reports to be identified.
- LastCheck: This configuration parameter is a date/time which stores the end date/time of the period a data integrity check was last queued for.
- LastSync: This configuration parameter is a date/time which stores the end date/time of the period a data synchronisation was last queued for.
- PurgeCompletedQueueItems: This configuration parameter is a Boolean which enables or disables the deletion of completed queue items. This configuration parameter is explained in more detail in section 6.5.4.

- SyncEnabled: This configuration parameter is a Boolean which controls the status of data collection for all client historians. If this is set to 'False' no data collection will be performed, even if data collection is enabled for a given data source.
- SyncIntervalMinutes: This configuration parameter is an integer which specifies the time interval in minutes at which data synchronisation operations will be queued. The default value for this parameter is 30 minutes.
- SyncWaitMinutes: This configuration parameter is an integer which specifies the time interval in minutes that the middleware will wait following the end of a given time block before queuing data synchronisation operations for that time period. The default value for this parameter is 10 minutes.

Configuration parameter values are set and retrieved using a .NET class created for this purpose. A ConfigParameter<T> object is created for each parameter, where T is the type of .NET object represented by the configuration parameter. The ConfigParameter<T> object provides a Value property which enables the configuration parameter value to be retrieved or set.

6.5 Transaction Queue

To ensure the reliable collection of data a queue system with integrated integrity checking is used as described in section 5.5.4. To ensure data collection reliability even after a power failure or unexpected application restarts the queue is stored in the middleware database in a table as described in section 6.3.4. Utilizing the database table for storage of all queue operations ensures no state information is kept in RAM or other volatile storage.

6.5.1 Queue Operations

Each item in the queue is required to have an operation specified to indicate to the middleware what action is required. This section details all the operations that can be queued for processing by the middleware.

- GetData: The GetData operation is used to queue a data synchronisation for a single tag with the corresponding client historian. GetData operations will always have a StartTime and EndTime corresponding to the beginning and end of the time block to be synchronised.
- CheckData: The CheckData operation is queued when a data integrity check needs to be performed for a given tag. CheckData operations will always have a StartTime and EndTime corresponding to the beginning and end of the time block over which to perform the integrity check.

- ReCheckData: The ReCheckData operation is queued after a CheckData operation has failed and the data for the time period for the tag has been resynchronised with a new GetData operation. The ReCheckData will perform another integrity check against the client historian. The operation will always have a StartTime and EndTime corresponding to the time period that originally failed the data integrity check.
- PurgeData: The PurgeData operation is queued after a ReCheckData operation has failed. The PurgeData operation will remove all data for the specified tag over the given time period from the middleware database so it can be fully resynchronised from the client historian.
- ReCheckDataPurge: The ReCheckDataPurge operation is queued after a PurgeData and GetData have been performed for a given tag over a specified time interval. The ReCheckDataPurge will perform another integrity check against the client historian and send a notification to the company operations staff on failure. The operation will always have a StartTime and EndTime corresponding to the time period that failed the original CheckData and subsequent ReCheckData operations.
- CleanData: The CleanData operation is used to perform data compression after a CheckData, ReCheckData or ReCheckDataPurge operation has succeeded. The compression will remove any repeated samples from the middleware database to save space, and will always have a StartTime and EndTime corresponding to the integrity checked data.
- CheckTagExists: The CheckTagExists operation is periodically queued for every tag enabled for data collection. When executed it ensures the specified tag is still available in the client historian and will generate an alert to the company operations staff if the tag is not found. The CheckTagExists operation does not require a StartTime or EndTime value to be specified.
- SyncTags: The SyncTags operation is periodically queued for each enabled data source to synchronise the list of tags available in the client historian with the tag table in the middleware database. Existing tags are also updated if the name or other available details have been changed in the client historian. The SyncTags operation does not require a TagID, StartTime or EndTime value to be specified.

6.5.2 Operation Status

To track the current state of a given queue operation an integer representing the status of the item is used. These statuses are detailed in this section.

- STATUS_WAITING (0): The waiting status with an integer value of 0 is given to all new items added to the queue. An item with this status has either been recently added to the queue or has failed processing previously and needs to be attempted again.
- STATUS_COMPLETE (1): The complete status with an integer value of 1 identifies queue items that have been successfully processed. Any queue items with this status can be removed from the queue if desired.
- STATUS_TEMPDELAY (2): The temporary delay status is assigned to queue items after a failure during processing. Setting the failed item to temporary delay allows subsequent queue items to be removed from the queue and be processed despite the earlier failure. When the queue has had all waiting items with a status of 0 processed the queue items with a status of temporary delay will be updated to a status of waiting so an attempt is made to process them again.

6.5.3 Operation Priorities

To allow queue items to be prioritised each row in the queue table has a corresponding priority expressed as an integer. When the next queue item to be processed is being selected the items with lower priorities are always processed first. The middleware queues all automatic data collection operations with a priority of 1, and any backfill operations queued via the common interface for historic data are queued with a status of 5. This ensures the automatically queued collection for current data will always occur before any backfills that have been queued externally.

To allow for future development the priority of 0 has been reserved for operations that may need to pre-empt automatic data collection. The priorities 2 through 4 and 6 onward are not currently used so are also available for adding additional operations that may need to be implemented during further development. It is possible the common interface could be extended to give consumers of the interface the ability to specify priorities for submitted backfills. The priorities accepted via the interface may need to be restricted to prevent queue items that would delay automatic data collection being submitted.

6.5.4 Queue Pruning

Without a process for pruning completed queue operations the queue table would grow unnecessarily large. While keeping historic queue items can aid in debugging issues it is generally not desired in a production environment. When the PurgeCompletedQueueItems configuration parameter is set to ‘True’ the middleware will remove queue operations as they successfully complete. Should the historic queue items need to be retained for debugging

purposes the configuration parameter can be set to 'False' which will cause completed queue items to be kept until manually removed.

6.6 Main Application Loop

The main application loop forms the core of the middleware as it controls the addition of operations to the queue as required by the configuration database. When running as a service the application loop is started as a thread when the service is launched. When starting the thread performs initialization of the service and spawns a worker thread as described in section 6.7. The thread continues to run until a request to stop the service is received.

The main application loop is implemented as a while loop. On each execution the loop will run and check for any operations that need to be inserted in the queue. The main thread will then be blocked until either the start of the next minute or a request to stop the service is received. This results in the loop executing approximately once a minute when the service is running normally.

The first check undertaken when the loop executes is to check if a data synchronisation is required. To perform this check the current time is offset by subtracting number of minutes in the value stored in the SyncWaitMinutes configuration parameter. The number of minutes past the hour of the offset time is then divided by the desired sync interval stored in the SyncIntervalMinutes configuration parameter. If the remainder is zero then a data sync needs to be queued. In this case the middleware queues a data synchronisation for all enabled tags from the time in the LastSync configuration parameter until the calculated offset time. If this data synchronisation is successfully queued the value of LastSync is updated to be the calculated offset time. To ensure the middleware contains an up to date list of tags available in client historians a synchronisation of tags is queued following all data synchronisations.

The middleware then checks if a data integrity check is required. The application loop contains a Boolean variable that is set to true if the date has rolled to the next day since the last loop execution, which signifies an integrity check is due. If the check is due the total number of minutes elapsed for the day is compared to the value of the CheckWaitMinutes configuration parameter. If the number is greater or equal a data integrity check is queued for all enabled tags, and the check due flag reset. The integrity check is queued from the time in the LastCheck configuration parameter until the end of the previous day. If this occurs successfully the LastCheck configuration parameter is updated to the date and time at the end of the previous day.

The final check made by the main application loop is to check if a heartbeat status update needs to be sent to the proprietary in-house monitoring system. These updates are sent every five minutes, with alerts being sent to administrators if a given process has not checked in for more than ten minutes. The time of the last status update is stored in RAM as it does not need to be retained between service restarts. If the status update is required a call to an external library is made which performs the status update via an HTTP request.

The application contains a Boolean variable with the desired status of the service. The value of this variable is set to true at service start-up and only changed to false if a request to stop the service is received. If the service is still required to be running after the loop has completed execution the thread is blocked using an EventWaitHandle until next execution. If the service has been requested to stop the EventWaitHandle will be signalled causing the loop to exit immediately. The service will then wait for the worker thread to join before stopping.

The main application loop contains an exception handler. In the event that an exception is generated during the loop execution an alert is sent to the system administrators. Generally exceptions in the main application loop are caused by unavailability of the SQL Server database. If the database is unavailable and the service is unable to queue new operations or access configuration parameters no data will be missed as the service will always queue operations beginning from the time the operation was last successfully queued.

The operation of the main application loop is shown in Figure 30.

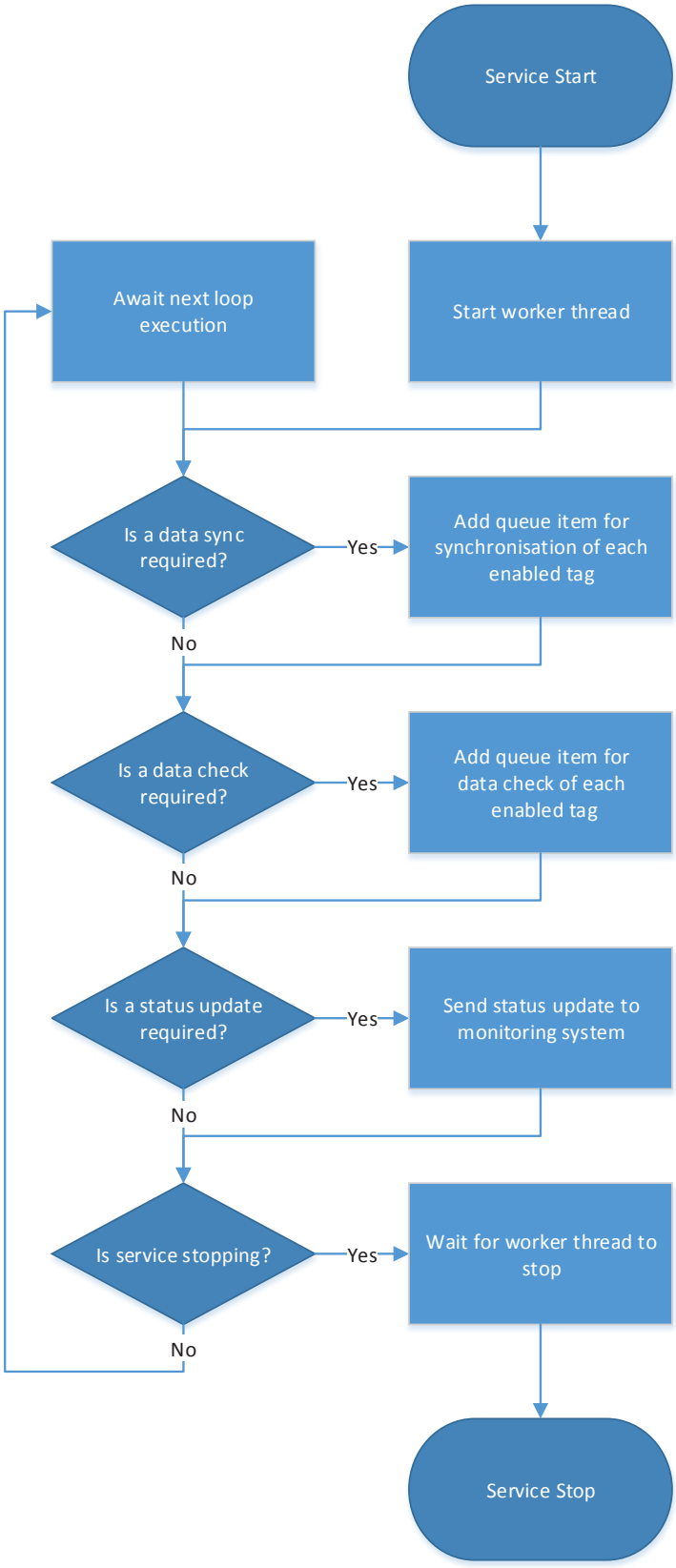

Figure 30: Main program loop

6.7 Worker Application Loop

The execution of queue operations within the middleware service is performed by the worker application loop. The worker application loop is started in a separate thread during the initialization of the main application loop.

The worker application loop is implemented as a while loop. When the loop executes it first attempts to retrieve the highest priority queue item with a status of STATUS_WAITING. To achieve this the query made against the STATUS_WAITING items in the queue database table first sorts the table by ascending priority, as the lower the priority integer the more urgent the queue operation as is described in section 6.5.3. The query then applies a secondary sort criteria of ascending QueueItemID and retrieves the first row of the result. This gives the next queue item as the oldest queue item of the highest priority.

If a queue item is available a switch statement is used to execute the appropriate code based on the value of Operation column. As the worker is a single thread all queue processing awaits the execution of the current item. If the queue item executes correctly the queue item is either deleted or given a status of STATUS_COMPLETE based on the middleware configuration. If the queue item does not successfully execute or an exception is thrown during execution the queue item is given a status of STATUS_TEMPDELAY. This allows the queue processing to continue with the failing item not blocking execution of the next STATUS_WAITING item.

When all items with a status of STATUS_WAITING have been completed any items in the queue with a status of STATUS_TEMPDELAY are reset to a status of STATUS_WAITING. This ensures the queue items are continually retried until they are processed successfully.

If no queue item is available with a status of STATUS_WAITING the loop uses an EventWaitHandle to block the worker thread until the handle is signalled by a request to stop the service, or a ten second timeout has elapsed.

In the event that the service is stopping the execution will break from the loop after the current item completes so the worker thread can join the main thread, allowing the service to exit.

The operation of the middleware worker loop is shown in Figure 31.

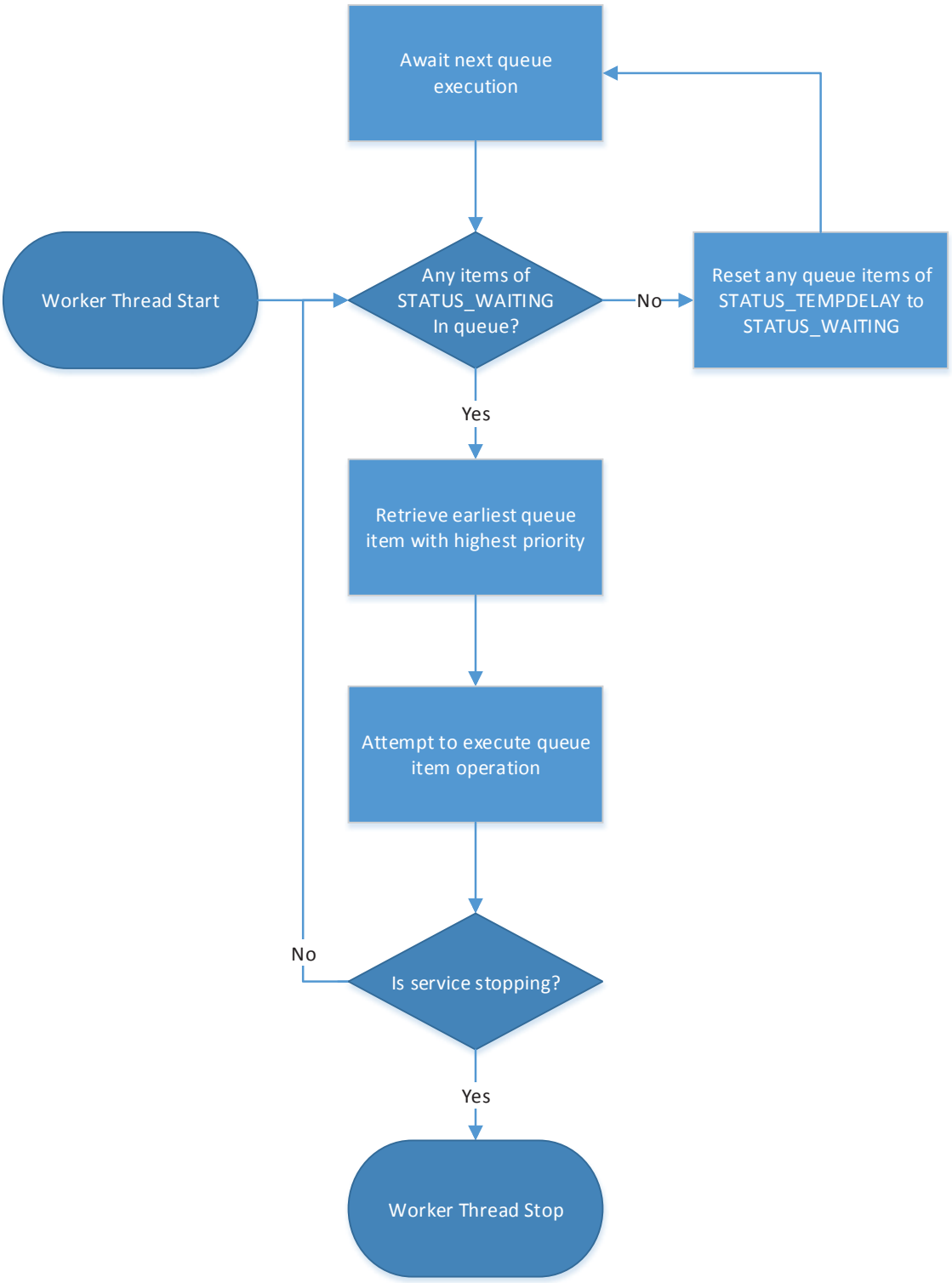

Figure 31: Middleware worker loop

6.8 Data Integrity Checking

To improve the reliability of data collection an integrity check is implemented in the middleware. The check is implemented as an extension of the method suggested in section 5.5.1. The integrity check occurs once per day and is queued by the application loop at a configurable number of minutes after the day has ended. Delaying the integrity check allows any data that may have been delayed in transit has a chance to arrive at the client historian.

To ensure reliable operation in the production environment the integrity check is designed to run unattended and is capable of solving some problems without operator intervention. This is achieved using a three-step checking process.

The first check performed by the middleware consists of a simple sample count comparison. The middleware requests from the historian the number of samples it contains for the time period being checked. This figure is then compared to the number of samples stored locally in the middleware database. If the number of samples matches the integrity check has passed and a data clean is queued for the tag for the integrity-checked time period.

If the integrity check fails a data synchronisation is queued for the tag for the time period. The DataSync operations are queued with ReCheckData operations following. As the ReCheckData operations are executed the same integrity check is performed with the number of local samples being compared to the count of samples in the client historian. If the integrity check succeeds a data clean is queued as in the first check.

If the second check fails it is possible there are erroneous samples in the local database that need to be removed. To do this a PurgeData operation is queued which when executed removes all data from the local database for the failing time period. The PurgeData operations are followed by DataSync operations, which are in turn followed by ReCheckDataPurge operations. As the DataSync operations execute the local database is repopulated with samples for the failing time period. The following ReCheckDataPurge operation then performs the third integrity check, once again comparing the number of local samples to the count of samples in the client historian. As previously a data clean operation is queued for the tag if the integrity check succeeds.

Failing the third integrity check leaves the middleware in a state it cannot resolve in an unattended manner. Using the proprietary monitoring system an alert is generated and sent to the system administrators so the issue can be investigated manually. In this case a data clean

operation is not queued as the raw data in the middleware database needs to be available so it can be compared to the samples in the client historian.

The diagram in Figure 32 demonstrates the extended data integrity checking process as is implemented in the middleware.

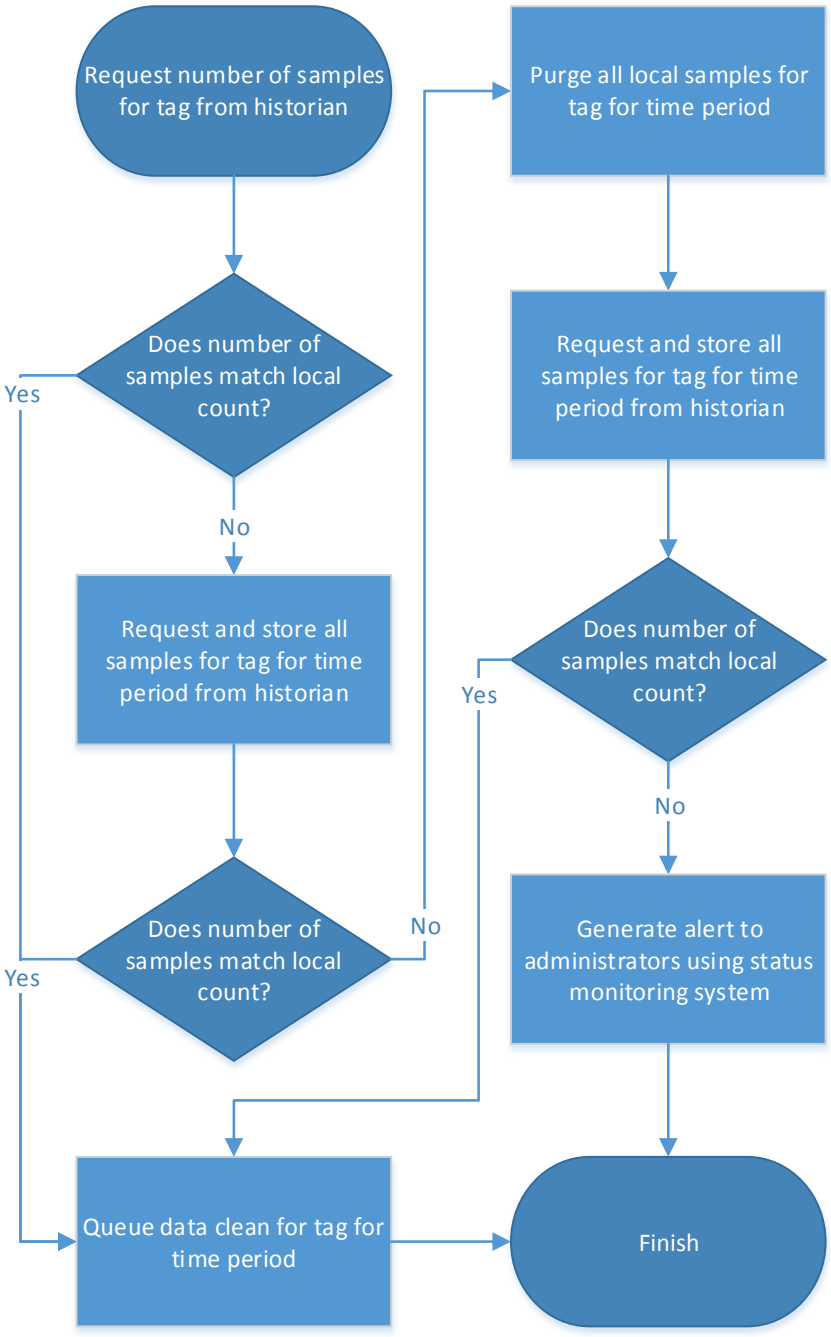

Figure 32: Extended data integrity checking scheme

6.9 Data Cleaning

To ensure optimal use of storage space a compression algorithm is implemented in the middleware. To ensure client data integrity is preserved the data compression is implemented through duplicate value removal as discussed in section 5.7.2. This ensures there is no loss of precision in the stored client data.

The duplicate data removal needs to be implemented as to not break the operation of the data integrity checking. The data integrity checking algorithm relies on counting the number of samples present in the local database and comparing that to the number available in the client historian. If duplicate values were removed from the data retrieved from the historian as it was inserted in to the middleware database the count of values available locally would be less than that in the client historian. As determined in section 5.7.2 approximately 71% of the data retrieved from the historian would be discarded. To prevent this the data cleaning operations are not scheduled by the middleware until after the integrity check for a tag over a given time block has completed successfully. After the success of the first, second or third integrity check a CleanData operation is scheduled for the tag for the same time period that passed checking.

To perform the data compression the latest value preceding the compression time period is retrieved and stored in a variable VALUE. This ensures that if the first value of the compression time period is a duplicate it is removed. The code then iterates over the data for the compression time period, comparing the current value to VALUE. If the values match the current value is added to a list of samples to be removed. If the values do not match VALUE is updated with the current value and the compression continues to the next sample. After a list of samples for removal has been obtained the samples in the list are permanently deleted from the sample database table. An illustration of the data compression algorithm is shown in Figure 33.

As the CleanData operation is only scheduled automatically after integrity checks any backfill requests for historic data scheduled using the common interface will also have a corresponding CleanData operation appended to the queue.

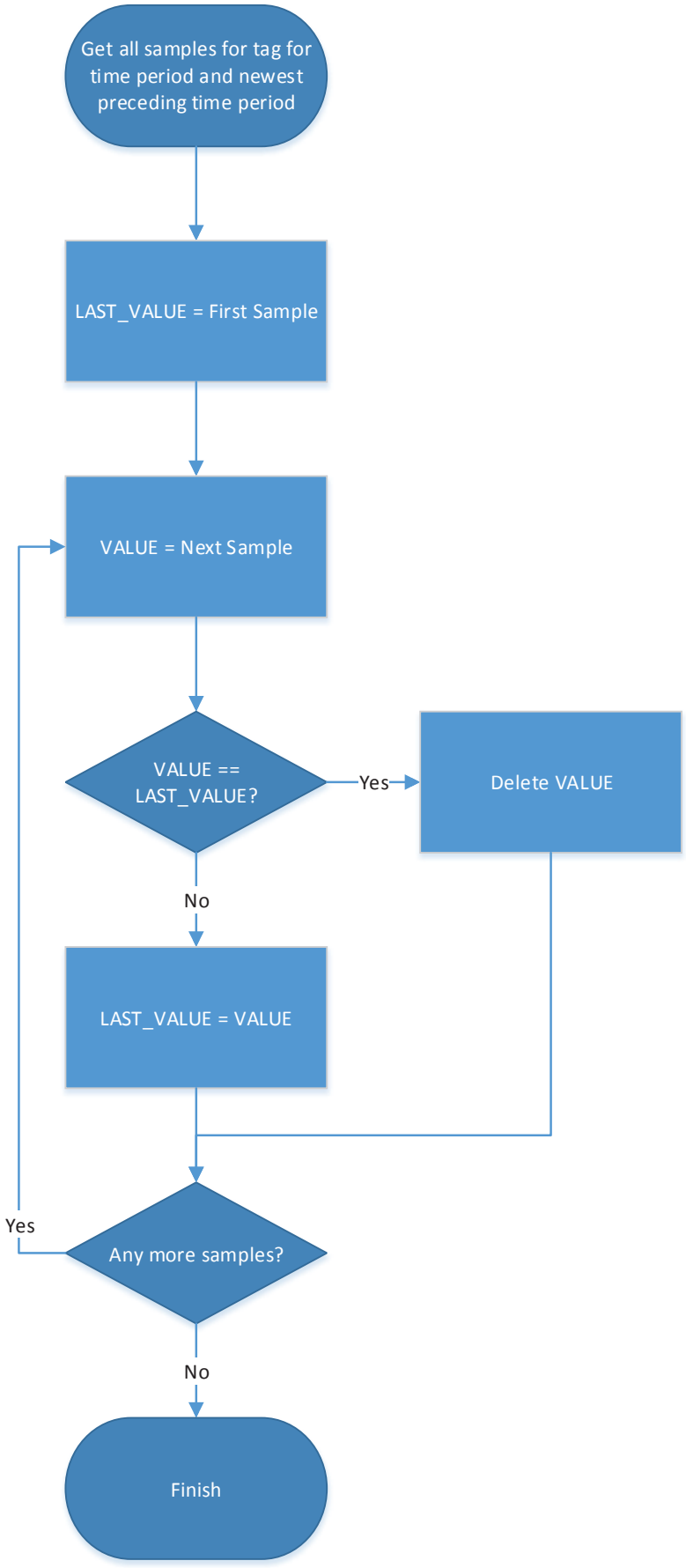

Figure 33: Data compression algorithm

6.10 Historian Data Provider Interface

To facilitate communication with client historians a collection of historian data providers were developed for use in the middleware. These data providers each use a custom interface similar to that developed for use in external applications as described in section 5.3 to communicate with the middleware worker thread. This interface is described in detail in this section.

The method signature for the historian data provider is implemented as a C# interface. Each historian data provider class inherits this interface providing a common set of methods for the middleware to utilize. The individual historian data provides are managed by a static class which will return the appropriate interface when requested based on the historian type string from the data source row.

6.10.1 Data Retrieval

The method used in the historian data provider interface is very similar to the common interface GetData method. The method accepts a database tag row, a start time and end time as inputs. The response is a data table containing columns with the sample time, sample value and sample validity. The request and response input and output is shown in Figure 34.

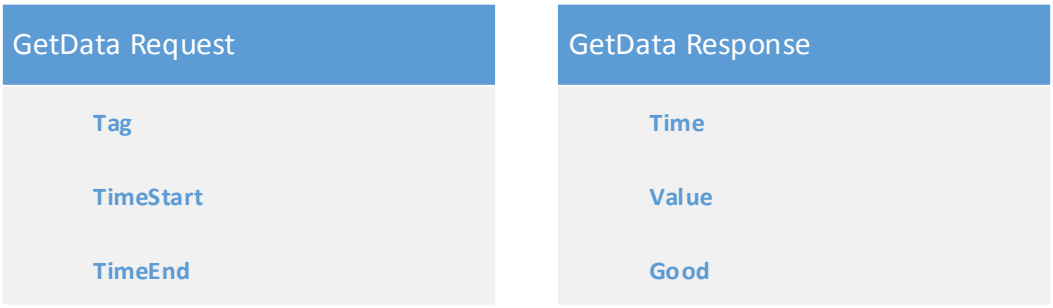

Figure 34: Historian data provider GetData request and respoonse

The middleware appends the tag information to each row in the GetData response before inserting the samples in to the middleware database sample table. The entire content of the response is inserted in the middleware database initially to allow integrity checking as described in section 6.8, with data compression taking place after integrity checking is complete.

6.10.2 Data Integrity Checking

To facilitate integrity checking the middleware requires a method of requesting the sample count for a given tag across a specified time period. In the historian data provider interface this is implemented in the GetCount method. The method accepts a database tag row, a start time and end time and inputs. The historian data provider requests the count of samples from the historian and returns it as an integer. The GetCount request inputs are shown in Figure 35.

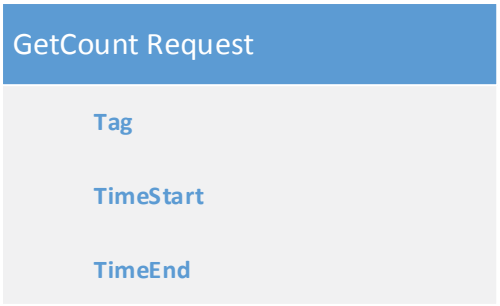

Figure 35: Historian data provider GetCount request

In the case that the historian does not support returning the sample count the data provider uses the GetData method to request all of the data for the time period but instead of returning a data table with samples returns only the row count as an integer. While requesting the data again is undesirable it does ensure the integrity check code can be used successfully for all historian data providers.

6.10.3 Tag Verification

To ensure a given tag is still available in the client historian the CheckTagExists method is called periodically. The method takes a tag row as an input and returns a Boolean value. If the tag exists the method will return true, otherwise false. The CheckTagExists inputs are shown in Figure 36.

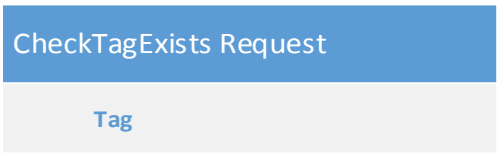

Figure 36: Historian data provider CheckTagExists request

If the call to CheckTagExists returns false an alert is sent to the company operations staff so the issue can be resolved manually. In the case that the tag has been renamed in the client historian resolving the issue would likely entail updating the middleware database with the new name manually. If the tag has been removed from the historian data collection would need to be manually disabled for the tag.

6.10.4 Tag Listing

Periodically the middleware will retrieve a list of tags from each client historian to ensure the local tag database is up to date. To request this information the middleware calls the GetTagList method of the historian data provider interface. The method accepts a data source table row as input and returns a data table containing the tag names and descriptions where available. The request and response input and output is shown in Figure 37.

Figure 37: Historian data provider GetTagList request and response

The middleware iterates over the tag list, updating local tag descriptions where they have changed and adding any new tags to the local database.

6.11 Historian Data Providers

Communication with historian products is handled by a historian data provider unique to each historian implementation. The data providers use the historian data provider interface as described in section 6.10 to communicate with the middleware worker thread and perform the appropriate translation of requests and responses sent and received from the corresponding client historian. The data provider for each supported historian is described in this section.

6.11.1 Citect Historian Data Provider

As Citect Historian uses an SQL Server database for data storage all communication is performed using the Microsoft .NET Framework Data Provider for SQL Server. The connection string stored in the middleware data source table is passed directly to the SQL Server data provider to initiate a connection to the Citect SQL Server.

The data for the middleware Tag table is mapped from the Citect Tags table. The middleware TagName column is mapped directly from the Citect TagName column, with the middleware Description column being left as a null value as is shown in Table 1.

Middleware Tag Column	Citect Tag Column
TagName	TagName
Description	NULL

Table 1: Citect Tag Mapping

The data for the middleware Sample table is mapped from both the Citect DigitalSamples and NumericSamples tables, with the mapping for both tables being identical. Citect stores the sample date as bigint which is converted to a UTC SQL Server datetime using the Citect provided ToDateUTC function. The sample value is mapped to the middleware Value column from the Citect SampleValue column. The Citect quality column contains an integer with 192 representing a good sample which maps to the middleware Good column as true, any other QualityID value will map as false. These mappings are shown in Table 2.

Middleware Sample Column	Citect Sample Column
Time	ToDateUTC(SampleDateTime)
Value	SampleValue
Good	QualityID == 192

Table 2: Citect Sample Mapping

Citect Historian does support storage of string samples, however these were not utilized by any of the council clients who used Citect Historian and therefore support was not required in the middleware implementation.

6.11.2 GE Proficy Historian Data Provider

GE Proficy Historian uses a proprietary database for sample storage. They provide an OLE DB driver which is capable of connecting to the historian to retrieve data using SQL queries. To integrate this OLE DB driver in the middleware C# service the Microsoft .NET Framework Data Provider for OLE DB is used. The connection string which is passed to the .NET Framework Data Provider for OLE DB includes the name of the GE Proficy OLE DB driver along with credentials that are passed to the GE Proficy OLE DB driver. Once connected SQL queries are executed in a similar manner to a SQL Server connection. The connection methodology for GE Proficy Historian is shown in Figure 38.

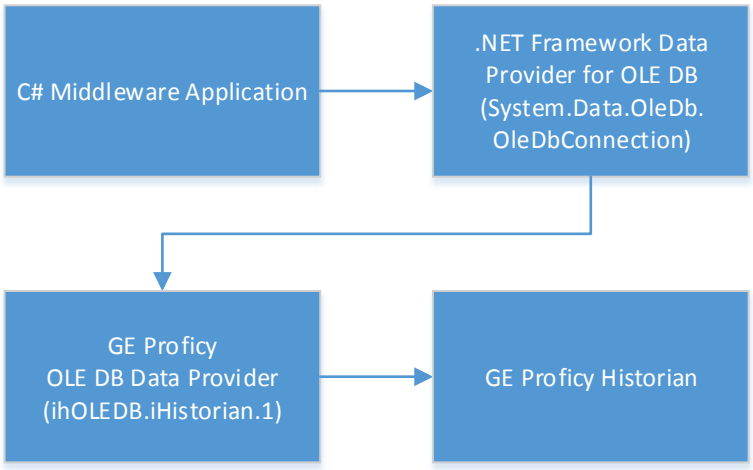

Figure 38: GE Proficy Connection Methodology

The data for the middleware Tag table is mapped from the Proficy ihTags table. The middleware TagName column is populated using the Proficy tagname column, with the middleware Description column being the Proficy description and engunits columns concatenated as is shown in Table 3.

Middleware Tag Column	Proficy ihTags Column
TagName	tagname
Description	description (engunits)

Table 3: GE Proficy Tag Mapping

The data for the middleware Sample table is mapped from the Proficy ihRawData table. When making queries against the historian query options can be specified using the WHERE clause. The middleware uses these options to ensure the data it retrieves is consistent. These options and the corresponding values are detailed below.

- samplingmode: This option is set to 'rawbytime' to ensure the historian passes back raw data that has not been interpolated to a fixed time interval. This ensures the precision of the data is retained.
- timezone: This option is set to '0' as this causes the historian to supply data with no timezone offset. As the middleware stores date and time data in UTC it is preferable that the data is obtained from the historian without a time zone offset.
- daylightsavingtime: This option is set to 'false' to ensure the historian does not apply any daylight saving time offset to the data it returns, and helps ensure the middleware gets a UTC date and time.

The timestamp column returned by the historian is mapped to the middleware Time column directly as it comes from the historian in UTC. The Proficy value column is mapped directly to the middleware Value column. Proficy returns a string detailing the quality of each returned sample. If this quality column value begins with 'Good' the sample is valid and the middleware Good column is set to true. Any other prefix on the Proficy quality column causes the middleware Good column to be set to false. These mappings are detailed in Table 4.

Middleware Sample Column	Proficy ihRawData Column
Time	timestamp
Value	value
Good	quality starts with "Good"

Table 4: GE Proficy Sample Mapping

During testing a memory leak was observed in the GE Proficy OLE DB driver. In the case that the Proficy OLE DB provider cannot connect to the Proficy historian the memory usage of the application will increase with the memory not being released after the connection attempt. If the connection issues persist this eventually causes the middleware service to run out of memory and terminate. This is mitigated in the production environment by setting the Windows automatic restart options for the service. When the middleware service exits unexpectedly

Windows restarts the service and queue processing continues. In future this could be further improved by spawning the data collection in a separate process which exists only for the lifetime of the queue item data collection.

6.11.3 Invensys Wonderware Historian Data Provider

Invensys Wonderware Historian uses a hybrid of SQL Server and a proprietary database for data storage. All access to the stored data is performed using the SQL Server interface, with modules implemented within the SQL database retrieving the data from the proprietary database if required.

The data for the middleware Tag table is mapped from the Wonderware Tag table. The middleware TagName column is populated from the Wonderware TagName column. The middleware Description column is populated from the Wonderware Description column. These mappings are shown in Table 5.

Middleware Tag Column	Wonderware Tag Column
TagName	TagName
Description	Description

Table 5: Invensys Wonderware Tag Mapping

The middleware Sample table is populated with data from the Wonderware History table. The Wonderware DateTime column maps to the Middleware Time column, and the Wonderware Value column maps to the middleware Value column. Wonderware provides a basic status indicator as a tinyint in the Quality column. When this column contains a value of 0 it is mapped as true in the middleware Good column. When the Quality column contains a value not equal to 0 it is assumed to be a bad sample and maps as false in the middleware Good column.

Middleware Sample Column	Wonderware History Column
Time	DateTime
Value	Value
Good	Quality == 0

Table 6: Invensys Wonderware Sample Mapping

During the development of the middleware negotiations were still underway with clients requiring Wonderware historian access, meaning no instance of Wonderware was available for testing. As described in section 5.2.6 a database was created containing replicas of the needed Wonderware tables. This database was used for testing the middleware integration with Wonderware historian.

6.11.4 QTech DATRAN Data Provider

QTech DATRAN uses SQL Server for data storage. As with previous SQL Server based historians the data can be accessed using the Microsoft .NET Framework Data Provider for SQL Server. The connection string stored in the middleware data source table is passed directly to the SQL Server data provider to enable a connection to the SQL Server containing the DATRAN data.

The data for the middleware Tag table is mapped from the DATRAN tblPoint table. The middleware TagName column is populated with the DATRAN strStation column concatenated with the strPoint column. The two columns are separated with a space-padded dash to ensure the station and point name can be easily read when required. The middleware description column is set to a null value as there is no suitable data available to populate it. The mapping for the middleware Tag table is shown in Table 7.

Middleware Tag Column	DATRAN tblPoint Column
TagName	strStation - strPoint
Description	NULL

Table 7: QTech DATARN Tag Mapping

The analog samples collected by DATRAN are stored in the tblAnalog table. The DATRAN dtTime column maps to the middleware Time column, though as DATRAN stores the date and time in local format it is converted by the middleware to UTC before being returned by the middleware data provider. The DATRAN fltValue column is mapped to the middleware Value column. As DATRAN doesn't differentiate the quality of samples the value in the middleware Good column will always be true. The DATRAN analog mapping is shown in Table 8.

Middleware Sample Column	DATRAN tblAnalog Column
Time	dtTime
Value	fltValue
Good	TRUE

Table 8: QTech DATRAN Analog Sample Mapping

DATRAN stores digital samples in the tblDigital table. Each row in this table has a start time, and optionally a stop time corresponding to the period the digital input was supplied a logic high signal. This results in a single row in the tblDigital table mapping to one row in the middleware sample table when a DATRAN sample has only a start time in the dtStart column, or two middleware samples when the DATRAN sample has both a start time in the dtStart column and a stop time in the dtStop column. In the first middleware sample the Time column is mapped to the DATRAN dtStart column, with the middleware Value column being set to 1 and the Good column being true. If the stop time is present the second middleware sample maps the Time

column to dtStop, with the middleware Value column being set to 0 and the Good column again being true. These two samples and the corresponding DATRAN tblDigital columns are shown in Table 9. As with the analog samples the middleware converts the local time returned by DATRAN to a UTC value.

Middleware Sample 1 Column	DATRAN tblDigital Column
Time	dtStart
Value	1
Good	TRUE

Middleware Sample 2 Column	DATRAN tblDigital Column
Time	dtStop
Value	0
Good	TRUE

Table 9: QTech DATRAN Digital Sample Mapping

As the time stored by DATRAN contains the local offset errors will be introduced during one of the daylight saving transitions. The date and time from the transition forward in which the local time advances an hour can be easily converted back to UTC time without error. However during the transition back in which local time reverts an hour the same time period in local time occurs twice. In New Zealand this occurs at 3 a.m. on the first Sunday in April, with the time period from 2:00:00 a.m. to 2:59:59 a.m. occurring twice (Satyanand, 2007). As this time period occurs twice it is not possible to determine the UTC values for samples during this hour as the offset information has not been stored.

6.11.5 Hilltop DataTamer Data Provider

Hilltop DataTamer uses a proprietary format for storage of sample data. To access this data they make available an Internet Information Services module which responds to data requests over HTTP. To make a request against Hilltop the middleware uses the .NET Framework HttpWebRequest class. The URL utilized in the request specifies the request type and any parameters needed. The data is returned as XML in the body of the HTTP response which is parsed by the .NET XmlReader class. The XmlReader class provides a means of iterating over the entire response in the order it appears in the XML document without having to load the entire document in memory.

To obtain tag information the middleware first requests a list of available sites from Hilltop. For each site Hilltop returns the middleware makes another request to retrieve a list of measurements that are available at the site. The middleware TagName column is populated with the Hilltop site name concatenated with the measurement name. The site and

measurement names are separated with a space-padded dash to ensure readability as is shown in Table 10.

Middleware Tag Column	Hilltop Mapping
TagName	Site Name - Measurement Name
Description	NULL

Table 10: Hilltop DataTamer Tag Mapping

When retrieving data from Hilltop the middleware makes a GetData request with the site name, measurement name and time range. Hilltop returns an XML document containing a <Data> element, inside which are a number of <E> elements. These <E> elements each contain a sample, with the content of the <T> child element containing the sample time in ISO 8601 format which is mapped to the middleware Time column, and the <I1> child element containing the sample value which is mapped to the middleware Value column.

Middleware Sample Column	Hilltop <E> Element Child
Time	<T>
Value	<I1>
Good	TRUE

Table 11: Hilltop DataTamer Sample Mapping

During the development of the middleware negotiations were still underway with clients requiring Hilltop data access, meaning no instance of Hilltop was available for testing. As described in section 5.2.8 a web application capable of emulating a Hilltop server was developed and used for testing of the middleware integration.

6.12 Connection Security

As the middleware has been operating in production with connections to SCADA historians hosted within council networks numerous discussions have occurred with council IT staff regarding connection security. In every case so far the council staff have been happy to use the connection and security methodology as described in section 5.6.3. The source address whitelisting with authentication is deemed to provide an adequate level of security for the type of data being transferred.

6.13 Middleware Deployment

To facilitate installation of the middleware as part of an automated process a MSI installer was built using a Visual Studio 2010 Setup project. The generated MSI file copies the files required by the middleware to the local disk and installs the middleware as a Windows service. The installer then sets the service failure options to allow Windows to restart the service in the event that it exits unexpectedly.

The installer has an unattended mode to allow the installation to proceed without user intervention. This mode is used by an internal company tool for both the initial deployment of the middleware and subsequent updates that may be required. When performing an update the installer stops the previously installed version of the service, installs the new version and restarts the service.

6.14 Queue Management Tool

To allow company support staff to perform simple maintenance of the middleware without requiring database access or knowledge of SQL Server a simple queue manipulation tool was developed. The tool allows data synchronisation, data cleaning or data synchronisation and data cleaning operations to be appended to the middleware queue. To operate the user selects either a single or all middleware data sources, a date range, a queue operation and a time block size. When the Queue button is pressed the operation is queued for all enabled tags in the given data source, with the queue item start date advancing by the specified interval length until the operation has been queued for the entire date range.

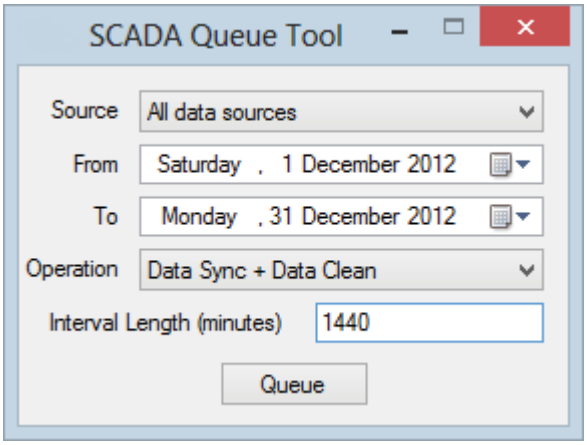

Figure 39: Middleware Queue Tool

The queue tool is included in the MSI installer and is installed alongside the middleware. An icon is created in the start menu allowing easy access for staff when attention is required. The most common use for the tool is queuing backfills for historic and data that may not have been collected by the client historian before the middleware performed integrity checking. Operations queued using the queue tool are given a priority of 5 to prevent them interfering with live data collection.

Chapter 7: Conclusions

The implementation and deployment of the middleware has been successful, with the middleware currently being used in production with numerous clients. All stated project objectives have been achieved successfully. This section details the results achieved with regard to each aspect of the project.

7.1 Conclusions

Interfaces were developed for five historian products during the course of this project, the five historians having being chosen based on a survey of New Zealand councils. Three of the interfaces were developed with access to client historian installations and therefore tested in a production environment providing reporting for councils. The first of these historian interfaces has been running and collecting data for approximately six months without issues.

Access to the final two historian products was not secured during the course of the project. To allow development to proceed a means of simulating each product was developed based on available manufacturer documentation. While the middleware implementation has been tested against these simulations no testing has been performed against the actual historian product. It is expected the middleware implementation will be compatible when tested with these products.

Through the course of the project no concerns have been raised by councils regarding the methodology used to connect to the historians they host on-site. All implementations so far have used a source address whitelisted connection with authentication at the historian level where available. While this has been sufficient so far other options including VPN connections are available should a higher level of security be required by any future clients.

The queue based system implemented in the middleware has worked without issue during the testing phase of this project. In a number of instances there have been connectivity issues between the middleware and client historians due to internet outages and planned maintenance. On every occasion the middleware has continued collecting data without any intervention when the internet connection returned.

Using daily integrity checking has been crucial to the success of this project. On numerous occasions the integrity checking system has queued additional data synchronisations where data has been delayed between the plant SCADA system and the client historian. This has happened without any staff intervention, providing a system that runs with minimal intervention.

On two occasions during the course of the project a client historian ceased logging data from plant SCADA systems. In both these cases the company support staff were able to use the queue manipulation tool to queue backfill operations for the time period over which the historian was not collecting data once the council had rectified the issue and backfilled the historian. Staff have also used the queue tool to queue backfills for substantial amounts of historic data.

Using SQL Server for the middleware data storage has been successful. The middleware installation has been simple as no additional software installation is required on the production virtual machines.

The performance of the database has proved exceptional, with the database easily handling the volume of data generated by the current clients. The database schema has not required any changes or updates.

The common interface assembly was readily integrated in to the production web interface and reporting code. The interface has provided all the necessary functionality needed for the production code to access the data it requires for reporting.

A number of agile development methodologies were used during the development of the middleware. While the list of target historians was assembled early in the project the process of negotiating access in some cases took time and in other cases was unsuccessful. The development schedule was quickly altered as access was made available to new client historians. Towards the end of the project when it became apparent access would not be available to two of the historian products emulation of the historian interfaces were quickly produced to allow work to continue.

To allow support for new historians and feature enhancements to be quickly introduced to the production environment releases were deployed internally in the test environment most weeks. After a short period of testing these releases were deployed to the production environment to give clients access to their data before development of the middleware had finished.

7.2 Future Enhancements

While the middleware is performing well at present there are a number of potential improvements that could be implemented given additional time. These improvements are detailed in this section.

7.2.1 Parallel Worker Threads

At present the middleware can only process one operation at a time as it utilizes only a single worker thread to perform queue operations. In the case that the middleware is connecting to multiple historians it would be possible to spawn one worker thread for each data source. While data collection speed has not been an issue in production thus far this increased parallelism may be necessary as larger councils begin using the service.

If the number of worker threads is increased a dispatcher may be required to ensure the same queue operation is not attempted by multiple threads. This may require additional column to be added to the queue database to mark a queue item as in progress, which when set would prevent the queue item being taken by another worker thread.

7.2.2 Queue Item Failure Count

With the present implementation it is possible for a queue item to remain queued indefinitely. If a failure occurs when processing a queue operation the item is given a status of temporary delay causing the item to be ignored until all other queue items have been successfully completed or delayed for the next queue execution. While this ensures a failing item cannot block the queue the failing item will still be attempted every queue execution. If the failure is caused by a timeout or other long process this could cause substantial delay at the start of queue processing – especially if there are a large number of failing items.

The queue table could be extended to include an integer count of failed executions. The value of the column would be incremented by one every time execution of the item failed. When this value increased above a predefined threshold the queue item could be set to a permanent delay status with the queue item remaining in the queue, but with no attempts being made to execute it. An alert could be sent to operations staff who could resolve the problem causing the failures and then reset the queue item status, allowing it to be executed again.

7.2.3 Self-Service Backfill Tools

Currently any backfill operations of historic data need to be added to the queue by company staff using the queue manipulation tool. Ideally this operation should be accessible to end users via the web portal so long as safeguards can be put in place to prevent an excessive quantity of items being queued.

Investigation would be needed to create appropriate limitations – these could include limiting the date range allowable for backfills, restricting the queue item time block size based on date

range or preventing new backfills being queued if there are already a large number of pending backfill operations.

To provide an indication of progress a display could be implemented showing the number of queue items currently awaiting processing. This could include showing the end user the date range any queued backfill items to allow them to see the current date period being executed.

7.2.4 Data Collection Processes

During development memory leaks have been observed in the GE Proficy Historian OLE DB connector when the historian is unavailable. At present a large number of subsequent failures connecting to a Proficy historian can cause the application to terminate as memory available to the middleware service on the host system is exhausted.

To prevent this the data collection for some or all historians could be spawned by the middleware in a separate process. As the OLE DB object would be hosted in a separate process that exits on completion of the assigned task any memory leaked would be reclaimed when the process exits.

To achieve this the queue item details or ID could be passed to the worker process as command line arguments as it launches. The worker thread in the middleware would wait while the worker process attempted the queue item. The worker process would update the queue table with the success or failure of the operation, and if successful insert any data collected directly in to the middleware database. On the worker process exiting the worker thread would continue execution.

7.2.5 Pass-Through Data Access

As the common interface applications use to access data is very similar to the data provider interface it would be possible to implement a pass-through mode on a data source or tag basis. When pass-through is enabled for a given data source or tag no it would not have any data collection queued during the normal synchronisation process. Instead when a request for data was received the middleware would select the appropriate historian data provider and pass it the request to be sent directly to the historian. When the historian returns the data it would be sent directly back to the client via the common interface.

To achieve this the method used by the common interface to communicate with the middleware would need to be changed. At present the common interface reads directly from the SQL Server database used by the middleware. In the case of a pass-through data source or tag no data would be present in the middleware database, so nothing would be returned by the common

interface. The common interface likely would need to communicate with the middleware service using a TCP socket with requests and responses being processed by the middleware service instead of the database server.

References

Allen, S. (2004). *SQL Performance Tuning using Indexes*. Retrieved from OdeToCode LLC: http://odetocode.com/Articles/237.aspx

Edberg, D., Ivanova, P., & Kuechler, W. (2012). Methodology Mashups: An Exploration of Processes Used to Maintain Software. *Journal of Management Information Systems, 28*(4), 271-303.

FreeTDS Project. (2011). *FreeTDS*. Retrieved from http://www.freetds.org/

GE Fanuc Automation. (2006). *Proficy Historian OLE DB Provider.*

GE Intelligent Platforms. (n.d.). Proficy Historian 4.5.

GE Intelligent Platforms. (n.d.). The Advantages of Enterprise Historians vs. Relational Databases.

Invensys. (n.d.). Wonderware Historian 2012.

Mahnke, W., Leitner, S.-H., & Damm, M. (2009). *OPC Unified Architecture.* Berlin: Springer-Verlag.

Microsoft Corporation. (n.d.). *.NET Framework Data Providers (ADO.NET)*. Retrieved June 18, 2012, from Microsoft Developer Network: http://msdn.microsoft.com/en-us/library/a6cd7c08

Microsoft Corporation. (n.d.). *Encrypting Connections to SQL Server*. Retrieved June 20, 2012, from Microsoft Developer Network: http://msdn.microsoft.com/en-us/library/ms189067.aspx

Microsoft Corporation. (n.d.). *LINQ to ADO.NET*. Retrieved June 18, 2012, from Microsoft Developer Network: http://msdn.microsoft.com/en-us/library/bb397942.aspx

Microsoft Corporation. (n.d.). *Maximum Capacity Specifications for SQL Server*. Retrieved from Microsoft Developer Network: http://msdn.microsoft.com/en-us/library/ms143432.aspx

Microsoft Corporation. (n.d.). *SQL Server Native Client (ODBC)*. Retrieved June 18, 2012, from Microsoft Developer Network: http://msdn.microsoft.com/library/ms131415.aspx

Middleton, E. (2011). *Myths about Historians*. Invensys.

OPC Foundation. (2013). *OPC Tech Info - OPC Historical Data Access Specification*. Retrieved from http://www.opcfoundation.org/Default.aspx/04_tech/04_spec_hda.asp?MID=AboutOPC

Satyanand, A. (2007, July 6). *New Zealand Daylight Time Order 2007.* Retrieved from New Zealand Legisaltion: http://www.legislation.govt.nz/regulation/public/2007/0185/latest/whole.html

Schnieder Electric Industries SAS. (2011). *CitectHistorian Technical Overview*.

Shore, J., & Warden, S. (2008). *The Art of Agile Development.* O'Reilly Media.

Troelsen, A. (2007). *Pro C# with .NET 3.0.* Apress.

Vasyutynskyy, V. (2007). Towards Comparison of Deadband Sampling Types. *IEEE International Symposium on Industrial Electronics.* Vigo, Spain.

Zolotová, I., Flochová, J., & Ocelíková, E. (2005). Database Technology and Real Time Industrial Transaction Techniques In Control. *Journal of Cybernetics and Informatics 5*, 18-23.