An Evaluation of Software Connection Methods for Heterogeneous Sensor Networks

M. Hammerton, J. Trevathan, T. Myers, W. Read

Abstract—The transfer rate of messages in distributed sensor network applications is a critical factor in a system's performance. The Sensor Abstraction Layer (SAL) is one such system. SAL is a middleware integration platform for abstracting sensor specific technology in order to integrate heterogeneous types of sensors in a network. SAL uses Java Remote Method Invocation (RMI) as its connection method, which has unsatisfying transfer rates, especially for streaming data. This paper analyses different connection methods to optimize data transmission in SAL by replacing RMI. Our results show that the most promising Java-based connections were frameworks for Java New Input/Output (NIO) including Apache MINA, JBoss Netty, and xSocket. A test environment was implemented to evaluate each respective framework based on transfer rate, resource usage, and scalability. Test results showed the most suitable connection method to improve data transmission in SAL JBoss Netty as it provides a performance enhancement of 68%.

Keywords—Wireless sensor networks, remote method invocation, transmission time.

I. INTRODUCTION

THE performance of a distributed application is largely influenced by the speed at which it exchanges messages. It becomes even more important when real-time processing of data streams is required. The velocity of the message exchange (or *transfer rate*), is a common issue as stream-based distributed applications are found in several types of businesses. Example applications include financial services, network and infrastructure monitoring, manufacturing, and sensor networks (SNs). Transfer rate latencies as little as one second can have detrimental effects on these applications [1].

Transfer rate problems can be due to software, hardware, or network related deficiencies. For example, the bandwidth of the network can be insufficient or the server can be short of resources. If the problem is software related, it may refer to the message size being too big, or the message-parsing taking too long. Furthermore, the connection methods (CMs) used by the distributed system for handling the message exchange may introduce delays that negatively affects performance.

This problem has occurred in the Sensor Abstraction Layer (SAL) SN application [2]. SAL is a *middleware integration*

M. Hammerton and W. Read are with the School of Engineering and Physical Sciences, James Cook University, Townsville, Queensland, 4811, Australia (phone: 6107 4042 5880; e-mail: Mark.Hammerton@my.jcu.edu.au, Wayne.Read@jcu.edu.au).

J. Trevathan is with the School of Information and Communication Technology, Griffith University, Brisbane, Queensland, 4111, Australia (phone: 6107 3735 5046; e-mail: j.trevathan@griffith.edu.au).

T. Myers is with the School of Business (Information Technology), James Cook University, Townsville, Queensland, 4811, Australia (phone: 6107 4781 6908; e-mail: Trina.Myers@jcu.edu.au).

platform for abstracting sensor specific technology to provide a user-friendly way of integrating heterogeneous types of sensors. SAL is employed by the Davies Reef SN project [3] and the SEMAT project [4], [5]. It uses RMI as its CM for providing a client with data from one or more sensors. However, RMI introduces delays while streaming data from the sensors to the client, which limits SAL's effectiveness. To improve SAL's performance, this paper concentrates on exchanging the CM rather than optimizing the processes in SAL.

Numerous CMs exist for use in Java-based applications. Examples are socket-implementations, Web Services, MPJ Express, Ibis IPL, Java Fast Sockets, and CORBA2, which are already partly the subject of performance studies [6-9]. However, the focus of this paper is not to give an analysis into the performance characteristics and underlying mechanisms of CMs, but rather to identify which CMs will provide an immediate performance improvement to a wireless sensor network application. Analysis for the causes for the performance enhancement is beyond this paper's scope. Here, the main goals are as follows:

- Determine alternative CMs to RMI *for use in SAL*;
- Implement a suitable test environment;
- Test/evaluate the CMs and recommend one or more as an alternative to RMI; and
- Implement the alternative CM(s) in SAL and verify the performance improvement.

For the purposes of this study we examined Java-based connections from the Java New Input/Output (NIO) framework. We considered several CMs including Apache MINA, JBoss Netty, and xSocket. A test environment was designed to evaluate each respective framework for its suitability in SAL based on *transfer rate*, *resource usage*, and *scalability*. Results showed that all NIO frameworks outperformed RMI in terms of streaming data. Overall, the most suitable connection method for improving the data transmission in SAL is to replace RMI with JBoss Netty as it provides a weighted performance enhancement of 68%.

This paper is organised as follows: Section II provides a brief overview of SAL. Section III gives an introduction to the Java CMs under consideration. Section IV explains the design of the performance evaluation tests. Section V presents and analyses the results of the performance tests. Section VI implements and evaluates the best performing CM directly in SAL. Section VII provides some concluding remarks.

II. SENSOR ABSTRACTION LAYER

SNs are becoming increasingly essential as sensors are

introduced into more areas of life. However, lack of standards makes it difficult to integrate heterogeneous sensors in a single SN. Therefore, middleware is important to have, as it is able to manage all types of sensors. This middleware is often implemented specifically for one SN, which makes it hardware dependent. Changes in the network, such as adding a new sensor, lead to manipulating the middleware code.

SAL is a middleware integration platform, which provides a plug-in-based model where support for new types of sensors can be loaded to the running system via plug-in. The system automatically detects and configures new sensors, if permitted by the hardware and operating system (OS). It provides a unified interface to all sensors by abstracting the sensor specific features. This simplifies access to a SN and the management/control of its sensors [2]. SAL can be seen as a low-level software layer as it bridges a network of sensors with high-level applications or further middleware technologies.

SAL consists of two components, the *SAL client* and the *SAL agent*. The SAL client represents an interface either to the user via a user interface or to other applications. SAL implements the SAL agent *Application Programming Interface* (API) in order to provide the SAL agent's functionality. The API is grouped into the following three categories:

1. *Sensor Management* - Methods to manage the pool of sensors and including operations for enumerating, adding and removing sensors.
2. *Sensor Control* - Methods to report on a sensor's capabilities and control the streaming of the data.
3. *Platform Configuration* - Methods to adjust the platform, e.g., add support for a new sensor type.

Each category uses a different markup language. The Sensor Management methods use *SensorML* [10], which describes a sensor's configuration. The methods in the category Sensor Control use *CommandML*. The CommandML documents contain a list of commands which are supported by a sensor. The last category, Platform Configuration, uses *Platform Capabilities and Configuration Markup Language* (PCML). PCML documents contain information on the platform configuration in order to support a certain type of sensor technology.

The SAL agent implements the various features of SAL. It runs on a platform which is connected to the sensors and therefore is regarded as a *sensor gateway*. The connection between the platform and the sensors can be either direct using platform specific input/output (I/O) ports like USB or indirect by using wireless technology. The SAL agent manages the sensors which are directly connected and are found by the agent and the indirectly connected sensors it has been told of. The SAL agent consists of three layers: *Agent Layer*, *Communication Layer*, and *EndPoint Layer* (Fig. 1).

The *Agent Layer* is responsible for the communication with the SAL client. It receives messages, parses and forwards them to the underlying *Communication Layer* and sends the response back to the client.

The *Communication Layer* provides methods for managing

and controlling sensors. The managing methods are used to configure and set up hardware while the controlling methods translate a generic command into a sensor native command, which then can be transmitted to the sensor. The generic commands are provided in the SAL API. For translating the generic commands two sub-layers are used. The *Abstraction Layer* is an adapter layer where the generic commands are implemented. From here the sensor-specific methods are called, which are implemented in the *Protocol Layer*.

The *EndPoint Layer* is tightly coupled to the I/O ports available on the sensor gateway. It is responsible for transmitting native sensor commands produced by the Protocol sub-layer to the sensor, and data from the sensor is transmitted to the SAL agent. This layer's software code layer is normally included in the OS. SAL ensures it is available and configured correctly.
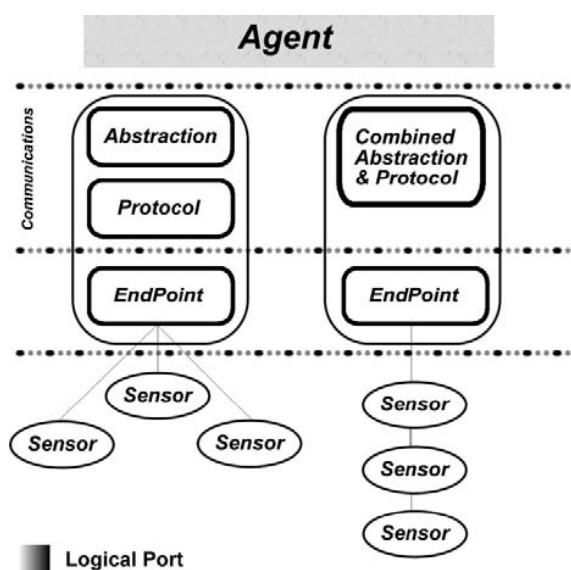

Fig. 1 SAL components and software layers

The sub-layers *Abstraction* and *Protocol* and the *EndPoint Layer* form a *Logical Port*. Each sensor is connected to a Logical Port that allows a specific client to control multiple agents. The logical port allows SAL to scale vertically in that an agent can control other agents in a hierarchical manner. The logical port treats an attached device as a data source, independent of whether it is a sensor or another agent. The ability to scale allows SNs of almost any topology or configuration.

The main objective of this project is to improve SAL's performance. The evaluation of the CMs needs to take into account the following three requirements:

1. Increase the *transfer rate* of messages between the SAL client and agent;
2. Improve the SAL agent's *resource usage* in terms of CPU utilisation and memory consumption; and
3. Improve the SAL agent's *scalability*.

III. CONNECTION METHODS

The following programming paradigms exist for implementing network communication between Java components [11]:

- *Sockets*: the communication is based on a one-to-one, duplex connection. The developer takes care of the data packing/unpacking and appropriate protocol usage.
- *Remote Procedure Calls (RPC) / RMI*: RPC enables calling remote methods as if they were local methods. The developer does not need to take care about the communication details. RMI is the object-oriented equivalent to RPC as it enables calling a method of a remote object.
- *Message Based Communication*: the communication is based on the exchange of messages with a predefined format. The communication details are hidden in an API.

A sensor gateway's resources can be limited. As this research aims to improve data transmission performance, only sockets are considered as an alternative communication method to RMI. Message based communication, e.g., SOAP3, would require sending and parsing of XML structures, which is performance hungry and increases the message size due to the XML overhead. Either the Java I/O API or the Java New I/O (NIO) API can be used to implement sockets. This paper uses NIO as it allows for a non-blocking implementation and therefore better performance and scalability.

A. Java Remote Method Invocation (RMI)

RMI enhances the creation of distributed Java applications, enabling the developer to invoke methods of objects running in a remote Java virtual machine. It provides an API for invoking remote methods with the same syntax/semantics as for local method calls. RMI is based on the principle of separating the definition and implementation of behavior by using interfaces (called *remote interfaces*).

A remote interface contains the definition of the methods being offered by the server as services for remote components. A server class implements the remote interface. Its instantiated objects (called *remote objects*) are registered with a naming-service, e.g., RMI-Registry. The clients connect to the naming-service and look up the references to the remote objects. They can then use the server's services as if they were local method calls. When using a method, its parameters are transmitted to the remote object, the appropriate method is invoked on the server and the result is transferred back to the client.

B. Java New Input/Output (NIO)

NIO introduced a range of features/improvements for I/O operations. These include buffer management; character set support, pattern matching with regular expressions, and scalable I/O operations for sockets and files.

NIO extends the I/O classes found in the *java.io* package. The developer chooses which approach to use depending on the application's requirements. Java I/O offers a stream-based approach using sockets for networking connections. However, this has scalability disadvantages due to its blocking nature.

The I/O operations behave in a synchronous way - a read operation waits until all the data is available and a write operation waits until all the data is sent. The thread performing the I/O is blocked and cannot proceed with other tasks. A separate thread is needed for each connection, resulting in a large number of resource consuming threads.

NIO provides *non-blocking* I/O operations, addressing the scalability problem faced with streams. This has the advantage that threads do not wait until an I/O operation is finished. A thread performing a write operation puts the data into an OS buffer and returns immediately. A thread performing a read operation instantly gets the available data rather than waiting until all data has arrived. Furthermore, NIO includes buffer management to improve performance issues. Buffers allow data to be transferred in bigger pieces than with streams, which mainly transfer data in small pieces, e.g., single bytes.

Since the introduction of NIO in J2SE 1.4 a variety of frameworks evolved, including Apache MINA, JBoss Netty, Grizzly, xSocket, and NIO Framework. These simplify the building of network applications by encapsulating the complexity of low-level I/O programming and extending the functionality of NIO concepts. Due to scope constraints, this paper restricts its attention to the following frameworks:

- *Apache MINA* (or MINA) provides an abstract, asynchronous, and event-driven API, which supports TCP and UDP. MINA (version 2.0.0) was chosen due to its popularity.
- *JBoss Netty* (or Netty) is also an asynchronous event-driven network application framework which provides a unified API for differing transport types, a flexible/extensible event model, and a customisable thread model. Netty (version 3.0.2) was chosen because it is the fastest out of all the aforementioned frameworks in tests conducted by T. Lee in [12].
- *xSocket* encapsulates low-level programming, has connection pool management, and connection timeout detection. xSocket (version 2.4.6) was chosen due to its popularity, and it is a lightweight easy-to-use framework.

IV. TEST DESIGN

A test framework was designed to reflect the communication between a SAL client and agent in order to evaluate the CMs. It is implemented as an independent application to prevent interference from background processes in SAL.

A. Test Criteria, Factors and Methods

The performance objectives outlined in Section II are to improve SAL's transfer rate, resource usage (CPU and memory) and scalability. Transfer rate and resource usage can be measured in a distinct dimension unit (e.g., bits-per-second or percentage). Scalability is measured by combining one of the two previous criteria with a factor, such as the number of clients. Other factors, which influence the criteria, include:

- *Data size* or the amount of data that is sent between a client and a server in a single message.
- *Data type* of a message (e.g., Integer, Byte, String, etc.).

- *Number of simultaneous clients* connected to the server.
- *Transmission mode* for how the requests are sent from the client to the server (e.g., blocking or non-blocking.).

A set of test methods were used to analyze the CMs regarding the aforementioned criteria. These included the *Transfer Rate* (TR), resource usage and scalability.

TR deals with the velocity of the message transportation between a client and a server. Two types of requests exist in SAL: synchronous and asynchronous. TRs for synchronous requests are measured using two different methods. *Round Trip Time* (RTT) measures the average time for sending a message from the client to the server and back. Two measuring points are needed on the client side: one before sending the request and the second directly after receiving the response. *Throughput* (TP) measures the number of messages that can be sent from the client to the server and back during a certain time span. This should be inverse proportional to RTT. The measuring point for counting the number of received messages is placed on the client side after receiving the response from the server.

Testing TR for asynchronous requests require a streaming-like approach. The *Streaming* test measures the number of messages received by a client in a certain time span. After receiving a request, a server continuously sends messages to a client until the client requests the server to stop. The measuring point is implemented on the client side by counting the number of messages received.

Resource usage refers to the server's resources (i.e., CPU utilization and memory consumption). The applied test methods are implemented on the server side.

Scalability is addressed by both TR and resource usage performed with different numbers of clients.

B. Test Scenarios

A test scenario defines the settings used in a test method. The definitions include the general test factors and test method-specific factors, such as time span or number of method calls. All test scenarios were performed at least three times to ensure that the collected data is representative. To reduce the interference of the Just-In-Time compiler, a large number of preliminary interactions were performed before the actual tests.

The data types/sizes are based on the values common in messages exchanged between a SAL client and agent. For example, String is often used for sending XML structures. It is included as data type with a size of 3408B. The maximum client number is 100 as a sensor gateway has limited resources. The transmission mode reflects both modes in which clients request data from the server. This can either be in a synchronous mode, where the client blocks until it receives the response from the server or in an asynchronous mode, where the client requests a data stream without blocking.

Each test method is performed with all possible values of the factor *Number of Clients*. The method RTT is performed with the data types Integer, Object, and String with their according data sizes. The transmission mode is synchronous.

In order to receive reliable results, the method is performed with 1000 method calls per client. The time is measured in nanoseconds.

The method *TP* has similar settings. It is performed with the data types Integer, Object, and String with their according data sizes and the transmission mode is synchronous. The time span for the message exchange is 60 seconds.

The method *Streaming* is performed using the data type Byte with the sizes 20KB, 40KB, and 80KB. The transmission mode is asynchronous and the time span for the client to receive messages is 30 seconds.

The test method *CPU Utilisation* and *Memory Usage* has two components, the client-server application and the monitoring tool. The transmission mode for the client-server application is synchronous and the data type is Byte. The data sizes are 10B, 100B, and 1024B. The monitoring tool reads the CPU utilisation and memory usage twice per second. The time span for exchanging the messages and the monitoring of the resources is 30 seconds. The previous scenario is also repeated whereby the transmission mode of the client-server application is asynchronous. The data type is Byte with the sizes 20KB, 40KB, and 80KB.

C. Test Environment and Implementation

The test setup used two PCs connected in a 100MBit/s LAN by a D-Link DSL-G604T router. One PC was used as a server (running Linux) whereas the second (Windows) was used for the client-application. The server simulates a sensor gateway running a SAL agent and the client simulates one or more SAL clients. The test framework was developed by using the Sun Java SE Development Kit Version 6 Update. The test platform is implemented as a client-server application written in Java. The client component manages the test methods, and the server processes the clients' requests. Both parts allow for easy integration of different CMs. A monitoring tool (BASH script) was used on the server side to record statistics on CM performance.

V. RESULTS AND ANALYSIS

Numerous preliminary tests were undertaken with each CM in isolation to each other. These tests analyzed each CM in terms of the criteria and factors described in the previous section. For brevity, the presentation and discussion of these results have been omitted. However, the overall results are used as input for the analysis that follows.

To analyze and evaluate the CMs against the background of improving SAL's performance, its specific requirements need to be taken into account. Each CM's test results are brought into relation with the results of RMI set to 100% as reference point. This highlights the differences between RMI and the tested NIO frameworks. For the final evaluation, a weighting of the conducted test methods and the criteria is applied. The weightings were chosen based on significance as follows: TR (50%); scalability (30%); and resource usage (20%). The overall scalability is composed of the scalability regarding TR (70%) and the resource usage (30%).
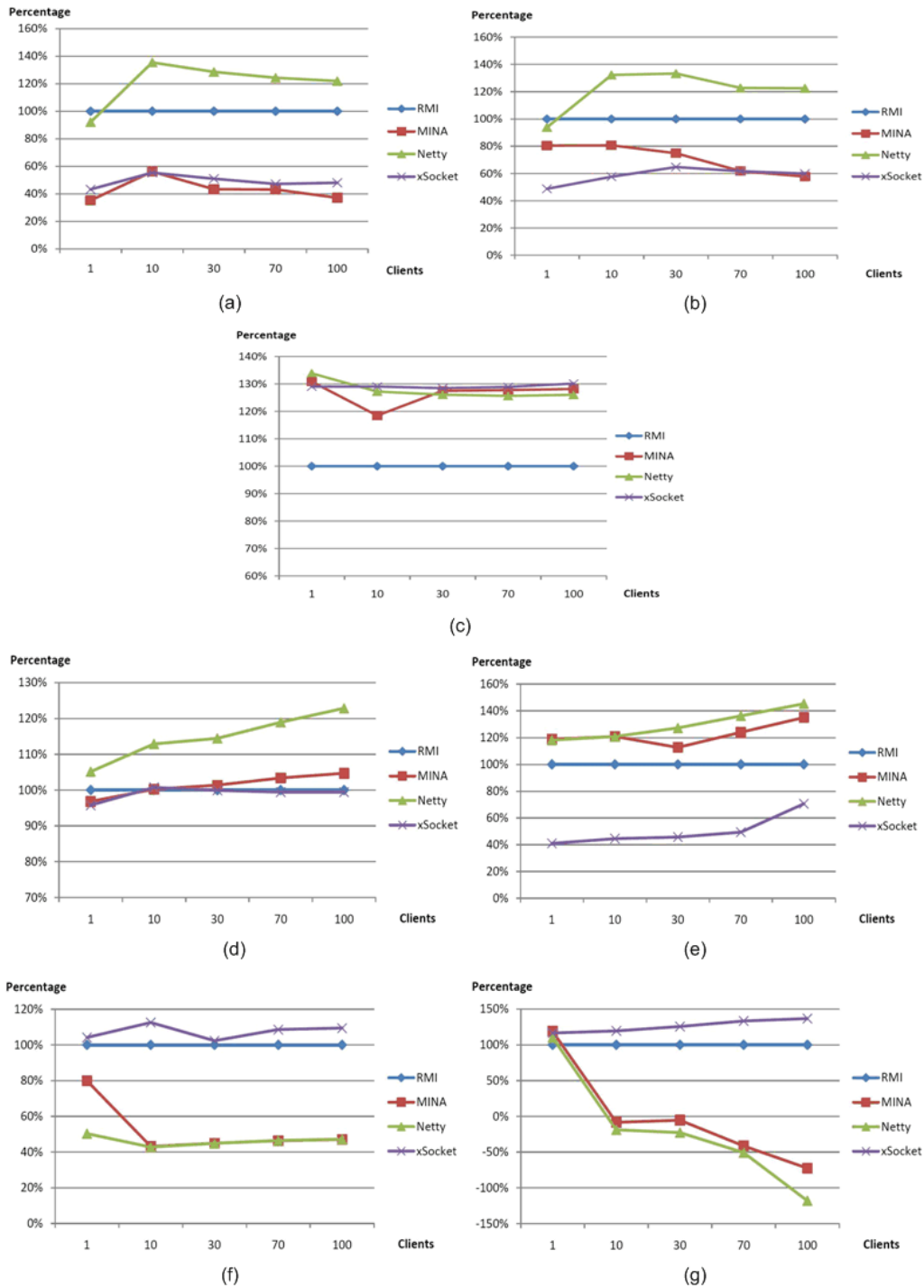
Fig. 2 Weighted Performance Analyses of Java Connection Methods with RMI as a Baseline (a) Round trip time tests (b) throughout tests (c) streaming tests (d) CPU utilisation tests for synchronous requests (e) memory usage for synchronous requests (f) CPU utilisation tests for asynchronous requests (g) memory usage for asynchronous requests

A test method's weighting is based on its importance in SAL. SAL mainly retrieves data using a streaming approach. Therefore, asynchronous mode is considered most important and gets a weighting of 70%. Consequentially the synchronous mode is weighted with 30%. Within the synchronous mode tests for determining TR, RTT and TP contribute equally and each get a weighting of 50%. The situation for CPU utilization and memory usage is similar and both get a weighting of 50%. Note that these weightings represent the authors' judgment of their relative importance to SAL and wireless sensor network

applications. Future work involves more formally defining the justifications behind each respective weighting.

A. Round Trip Time

The results for the RTT tests indicate that RMI is the best for one client, followed by Netty (see Fig. 2 (a)). Netty outperforms the other CMs for 10 or more clients (up to 35% better than RMI). Although the performance increase deteriorates with increasing client numbers, it is still 22% for 100 clients. MINA and xSocket are 45 to 65% less efficient than RMI.

B. Throughput

Netty outperforms the other CMs, being more than 30% better than RMI for 10 and 30 clients (see Fig. 2 (b)). Its performance decreases slightly for 70 and 100 clients, but with a gain of 23% it is still significantly better. In general, MINA's performance is not as good as RMI, with 19% for 1 client to 42% for 100 clients. xSocket is 50% worse than RMI, only showing a slight increase with increasing client numbers.

The cumulative results of the RTT and TP tests are strongly influenced by a distinct data type. While the String test results impact on the cumulative results in the RTT tests, their influence on the cumulative results of the TP tests nearly vanishes. The Integer tests have the biggest effect. This is apparent when observing MINA's results. Its overall bad performance for one client in the RTT tests is due to its String value results. Its performance in the TP tests is mediocre which is clearly owed to its outcome in the tests conducted with Integer values.

C. Streaming

The NIO frameworks perform similarly - approx. 30% better than RMI (see Fig. 2 (c)). The maximum increase is achieved by Netty for one client with 34%. Its performance then decreases slightly for 10 or more clients but is still at least 26% better. MINA has a steep decline for 10 clients decreasing to only 18% better, but then recovers to 28% better than RMI. xSocket is the best CM for 10 or more clients.

D. Resource Usage for Synchronous Requests

Netty scales the best in CPU utilization for synchronous requests (see Fig. 2 (d)). For one client it uses 5% less CPU and 23% for 100 clients. The remaining CMs have similar CPU utilization.

In terms of memory usage, MINA and Netty show a performance increase of 20% for one client, extending to 35% and 45% for 100 clients respectively (see Fig. 2 (e)). xSocket is generally worse than RMI, but scales a slightly better. The performance for one client is 60% worse, reducing to 30% for 100 clients.

E. Resource Usage for Asynchronous Requests

xSocket has a slight advantage in CPU utilization compared to RMI (see Fig. 2 (f)). MINA and Netty are the worst performers. xSocket's gain compared to RMI is between 2 and 3% for 30 and 10 clients respectively. MINA's performance is 20% more than RMI for one client, but decreases significantly for 10 or more clients. MINA and Netty use up to 57% more CPU resources than RMI.

All NIO frameworks have a better memory usage than RMI for one client (see Fig. 2 (g)). xSocket retains satisfactory performance with an increasing number of clients, but MINA and Netty show an increased memory usage. At its peak, Netty uses more than three times the memory as RMI for 100 clients.

F. Overall Comparison of CMs

To compare the CMs, all criteria are weighted as described in Section V. Each criterion is evaluated using a separate weighted table and the overall performance is combined in a final weighted table. RMI is used as a reference point, therefore it always has the value 100\%.

Table I lists the TR ratios with RMI as a reference point and the weighted ratios. MINA and xSocket clearly have a disadvantage for synchronous requests but this is offset by their good performance for asynchronous requests. Applying the weighting results in both CMs being a little better than RMI with a total value of 104% for xSocket and 109% for MINA. Netty has the best TR result of 122%. It has a slight disadvantage for synchronous requests but the best performance for asynchronous requests.

TABLE I
WEIGHTED TRANSFER RATE (WITH RMI AS A BASELINE)

	Ratio			Weighted Ratio		Total
	RTT	TP	Streaming	Synchronous	Asynchronous	
MINA	35.29%	80.51%	130.83%	17.37%	91.58%	108.95%
Netty	92.05%	93.94%	133.90%	27.90%	93.73%	121.63%
xSocket	43.25%	48.86%	129.03%	13.82%	90.32%	104.14%

TABLE II
WEIGHTED RESOURCE USAGE (WITH RMI AS A BASELINE)

	Ratio				Weighted Ratio		Total
	Synchronous		Asynchronous		Synchronous	Asynchronous	
	CPU	Memory	CPU	Memory			
MINA	96.82%	118.86%	79.82%	118.84%	32.35%	69.53%	101.88%
Netty	105.16%	118.11%	50.33%	110.10%	33.49%	56.15%	89.64%
xSocket	95.77%	40.98%	104.13%	116.52%	20.51%	77.23%	97.74%

TABLE III
WEIGHTED SCALABILITY (WITH RMI AS A BASELINE)

	Weighted Ratio				Transfer Rate	Resource Usage	Total
	Transfer Rate		Resource Usage				
	Synchronous	Asynchronous	Synchronous	Asynchronous			
MINA	17.12%	88.60%	33.53%	17.70%	75.51%	13.64%	90.15%
Netty	36.21%	89.46%	36.65%	9.19%	89.77%	13.10%	102.86%
xSocket	16.13%	90.39%	22.39%	81.79%	76.09%	29.77%	105.85%

Table II contrasts resource usage. xSocket's poor memory usage for synchronous requests is nearly leveled out by its performance for asynchronous requests leading to a total value of 98%. MINA's bad ratio for the CPU utilization is equalized by the better memory usage resulting in a total weighted value of 102%. Netty performs worst with a decrease of 10% due to its poor CPU utilization for asynchronous requests.

The scalability results show MINA performs the worst with a decrease of 10% compared to RMI (Table III). Netty scales well in terms of TR but have problems in the resource usage regarding asynchronous requests. It has a total value of 103%. xSocket scales best at 106%. The good performance for asynchronous requests offsets the bad performance for synchronous requests.

Table IV combines all three weighted criteria to give an overall evaluation. All NIO frameworks outperform RMI, with Netty being the best (110%), followed by xSocket (103%), then MINA (102%). As Netty had the most significant increase (10%), it was chosen for testing in SAL.

TABLE IV
FINAL WEIGHTING OF THREE TEST CRITERIA (WITH RMI AS A BASELINE)

	Transfer Rate	Resource Usage	Scalability	Total
MINA	54.48%	20.38%	27.05%	101.90%
Netty	60.81%	17.93%	30.86%	109.60%
xSocket	52.07%	19.55%	31.76%	103.37%

VI. IMPLEMENTATION IN SAL

A performance evaluation of Netty versus RMI when implemented in SAL is presented in this section. The tests for the synchronous mode are performed by requesting a list of sensors, while asynchronous mode tests request a data stream from a sensor. RMI is used as reference point.

For measuring TR, the synchronous and the asynchronous tests are set up as throughput tests meaning it is counted on the client side how many messages can be processed in a 30 second time span. The monitoring tool measures the resource usage (refer to Section IV). The CPU utilization and the memory usage are measured twice per second. For testing scalability, the aforementioned tests were conducted with 1, 10, 30, 70, and 100 clients. "Dummy" (or fake) sensors, which return a constant value, were used for both types of tests. This ensures the sensor returns consistent data that is always the same size and no sensor specific latencies influence the test results. The list requested from the clients during the tests in synchronous mode contained 28 sensors. Eight of these sensors were OS-related sensors, which are standard in SAL and 20 "dummy" sensors. The data for streaming in the asynchronous tests is requested from the "dummy" sensors.

The results of the synchronous tests indicate RMI's TP and TR for one client is a little higher than the Netty's TP. Netty's performance increases proportionally to number of clients. For 100 clients, Netty has an advantage of 37%. Netty's scalability regarding TR is clearly better in comparison to RMI.

The average frames-per-second rate per client while streaming data from dummy sensors for one client show a performance increase of Netty compared to RMI of 140%. For 10 and 30 clients, the increase is a little less but still nearly 90%. For 70 and 100 clients, it rises again and reaches a maximum of 166% for 100 clients.

The SAL agent's average CPU utilization while processing the request for the list of sensors shows that the usage for synchronous requests of both CMs is very similar. For one client, both CMs have a CPU utilization of about 76%. For 10 and more clients, the utilisation rises to about 95 to 98%. The difference between Netty and RMI reaches a maximum of 2% when handling 70 clients with the slight advantage for RMI.

The SAL agent's average memory usage while processing the request for the list of sensors shows that for 1, 10, and 30 clients, both CMs have a memory usage of about 14%. While Netty's memory usage stays constant, RMI increases to a maximum of 15.3%.

TABLE V
NETTY - WEIGHTED TRANSFER RATE FOR TESTS IN SAL (RMI AS A BASELINE)

Ratio		Weighted Ratio		Total
Synchronous	Asynchronous	Synchronous	Asynchronous	
97.30%	240.45%	29.19%	168.31%	197.50%

TABLE VI
NETTY - WEIGHTED RESOURCE USAGE FOR TESTS IN SAL (RMI AS A BASELINE)

Ratio				Weighted Ratio		Total
Synchronous		Asynchronous		Synchronous	Asynchronous	
CPU	Memory	CPU	Memory			
99.52%	99.59%	82.52%	96.61%	29.87%	62.70%	92.56%

TABLE VII
NETTY - WEIGHTED SCALABILITY FOR TESTS IN SAL (RMI AS A BASELINE)

Weighted Ratio				Transfer Rate	Resource Usage	Total
Transfer Rate		Resource Usage				
Synch	Asynch	Synch	Asynch			
35.59%	160.37%	30.34%	47.13%	140.20%	29.16%	169.37%

TABLE VIII
NETTY - FINAL WEIGHTING FOR THE TESTS IN SAL

Transfer Rate	Resource Usage	Scalability	Total
97.75%	18.51%	50.81%	168.08%

The SAL agent's average CPU utilization while streaming data from dummy sensors show that for one client Netty uses 17.5% more CPU than RMI. For 10 and more clients, both CMs have a similar behavior with both utilizing more than 93% of the CPU. RMI proves to have a slight but constant advantage compared to Netty.

The SAL agent's average memory use while streaming data from dummy sensors is similar to the Memory Usage tests for synchronous requests. The memory usage of Netty stays nearly constant at about 14% for all client numbers. RMI's memory usage increases from 13% for one client to 18% for 100 clients. This corresponds to an advantage for Netty of 24%.

The process described in Section V is used to evaluate the test results. Netty has a clear advantage in TR at 198% (Table V). However, Netty suffers at 93% in terms of resource usage (Table VI). When evaluating scalability, Netty has an overall positive value of 169% (Table VII). Table VIII combines the total values of all three criteria. Netty offers SAL a performance increase of 68% over RMI.

VII. CONCLUSIONS

This paper analyzed three different NIO frameworks as alternative CMs to RMI. The analysis was specifically *for use in SAL* with the goal of increasing message transfer in wireless sensor networks. A test platform was designed with a client-server application and a monitoring tool to test the CMs according to TR, resource usage, and scalability. The test methods used synchronous and asynchronous transmission modes. Results indicated that no particular CM outperforms the others in all tests - each had strengths and weaknesses.

A weighting was used for analyzing and evaluating the CMs, which focused on the TR and the tests conducted with asynchronous requests. The weighting was chosen to reflect the importance for SAL. The NIO frameworks were brought into relation with RMI by using the test results of RMI as reference point. Against the background of the defined weightings, Netty turned out to be the best CM. Then xSocket and MINA followed, where both also had positive total results compared to RMI. All NIO frameworks perform better than RMI.

As Netty appeared to be the most suitable replacement to RMI, it was implemented in SAL and its performance evaluated. In comparison to RMI, Netty improves TR and SAL's scalability significantly at the expense of slightly increased resource usage. The streaming capabilities of Netty were superior to RMI in SAL. Notably, the purpose of this paper was to identify a replacement CM to RMI. Analysing the underlying reasons for the performance increase is the focus of future work.

Other future work involves improving performance by using compression to reduce the data size being transmitted. Alternately, we will look into mechanisms for optimizing how the data is sent from the SAL agent to the client, or study if another thread model for data streaming would be more applicable. A further approach could consider the message delivery by pooling clients together who are interested in the same information to aggregate data transmissions. Finally, we could study how the performance changes when using UDP as a transport protocol rather than TCP, as UDP has a stateless nature, which is normally faster than TCP.

REFERENCES

[1] M. Stonebraker, U. Çetintemel, and S. Zdonik, "The 8 requirements of real-time stream processing," *ACM SIGMOD Rec.*, vol. 34, pp. 42-47, 2005.
[2] G. Gigan and I. Atkinson, "Sensor Abstraction Layer: a unique software interface to effectively manage sensor networks," in *Proc.3rd Int. Conf. Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP 07)*, Melbourne, Australia, 2007, pp. 479-484.
[3] C. Huddlestone-Holmes, G. Gigan, and I. Atkinson, "Infrastructure for a sensor network on Davies Reef, Great Barrier Reef," in *Proc. 3rd Int. Conf. Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP 07)*, Melbourne, Australia, 2007, pp. 675-679.
[4] J. Trevathan, H. Ghodosi, and T. Myers, "Efficient Batch Authentication for Hierarchical Wireless Sensor Networks," in *Proc. 7th Int. Conf. Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP 2011)*, Adelaide, Australia, 2011, pp. 217-222.
[5] J. Trevathan, I. Atkinson, W. Read, N. Bajema, Y. J. Lee, R. Johnstone, and A. Scarr, "Developing Low-Cost Intelligent Wireless Sensor Networks for Aquatic Environments," in *Proc. 6th Int. Conf. Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP'10)*, Brisbane, Australia, 2010, pp. 13-18.
[6] S. P. Ahuja and R. Quintao, "Performance evaluation of Java RMI: a distributed object architecture for Internet based applications," in *Proc. 8th Int. Symp. Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2000, pp. 565-569.
[7] W. R. Cook and J. Barfield, "Web Services versus Distributed Objects: A Case Study of Performance and Interface Design," in *Proc. IEEE Int. Conf. Web Services (ICWS06)*, 2006, pp. 419-426.
[8] C. Demarey, G. Harbonnier, R. Rouvoy, and P. Merle, "Benchmarking the round-trip latency of various java-based middleware platforms," *Studia Informatica Universalis Regular Issue*, vol. 4, pp. 7-24, 2005.
[9] M. Juric, I. Rozman, B. Brumen, M. Colnaric, and M. Hericko, "Comparison of performance of Web services, WS-Security, RMI, and RMI SSL," *Journal of Systems and Software*, vol. 79, pp. 689-700, 2006.
[10] G. Aloisio, D. Conte, C. Elefante, G. P. Marra, G. Mastrantonio, and G. Quarta, "Globus Monitoring and Discovery Service and SensorML for Grid Sensor Networks," in *Proc. 15th IEEE Int. Workshops Enabling Technologies: Infrastructure for Collaborative Enterprises*, 2006, Manchester, United Kingdom, pp. 201-206.
[11] S. Heinzl and M. Mathes, *Middleware in Java: Leitfaden Zum Entwurf Verteilter Anwendungen.* Wiesbaden: Vieweg+ teubner Verlag, 2005.
[12] T. Lee. (2011) Rapid Network Application Development with Apache MINA [Online]. Available: http://developers.sun.com/learning/javaoneonline/2008/pdf/TS-4814.pdf