An Object-Oriented Invocation Layer for the Java Message Service

Klaus Jank[1], Roy Oberhauser[1]

[1]Siemens AG, CT SE 2, Otto-Hahn-Ring 6,
81730 Munich, Germany
{klaus.jank, roy.oberhauser}@siemens.com

Abstract. New applications and software environments are increasingly distributed across a large or even unpredictable number of networked computing devices, require mobile and ad-hoc networking capabilities, and must integrate with more systems, all of which create greater demands on the middleware used to realize these systems. On the Java platform, RMI is a well-established paradigm, yet deficiencies become evident in particular with regard to scalability and remote invocation completability - which is the assurance that invocations are executed according to client and service expectations regardless of the state of the participants or the communication network. While the Java Message Service (JMS) addresses these deficiencies, it lacks the simplicity, explicit contracts, clear coupling, and strong typing of an object-oriented invocation paradigm. This paper will describe our Java Invocation Layer for Messaging (JILM), a(n) (a)synchronous invocation layer to support object-oriented invocations while leveraging the unique distribution and QoS properties that JMS provides.

1 Introduction

Distributed Java computing applications have often relied on RMI or RMI-IIOP [21] (referred to as RMI in this paper) because of its object-oriented paradigm, simplicity, and wide availability. However, changing environments and demands, such as a larger and often unpredictable number of networked computing devices (e.g., internet, embedded, and pervasive computing), greater mobility and ad-hoc networking (e.g., P2P, wireless), and increasing inter-system integration (e.g., intranets, B2B) cause RMI's deficiencies to become apparent, in particular scalability and remote invocation completability.

Scalability challenges occur when addressing an unpredictable number of participants or long duration invocations due to RMI's synchronous invocation model. The advantages of asynchronicity for scalability have been investigated, e.g. for CORBA in [1]. However, there are situations where it is desirable to be able to choose the appropriate mechanism in an invocation: asynchronous invocations – to fulfill scalability requirements, or synchronous invocations – to address programming complexity or to perform short duration invocations. Thus support for both invocation models is desirable.

Moreover, if a client makes an asynchronous invocation, this should not require the service to have to support the additional complexity of asynchronicity. But as system integration increases, more services rely on and utilize other services (i.e. service chaining), where the asynchronous invocation model may be preferable[1] for the service, such as has been argued for middle-tier servers in Eckerson [5] and Deshpande [4]. Consequently, the client and service invocation models should be decoupled, supporting independent client and service usage.

With regard to remote invocation completability in our scenarios, the following invocation properties, missing in RMI, become important: Time-Independent Invocations (TIIs), Location-Independent Invocations (LIIs), group invocations, batch invocations, interim or partial results, and Quality-of-Service (QoS). These will be described below.

Since the simultaneous availability of invocation participants cannot always be guaranteed, TIIs are desirable to allow the separate parts of the invocations to be stored and forwarded when each participant becomes independently available. This decouples the client and server lifetimes. The CORBA Specification [16] includes support for TIIs.

RMI invocations rely on object references which can change in dynamic systems, causing destination-specific, point-to-point invocations to fail. Similarly, the issue with Inter-Operable References (IORs) is known for CORBA invocations [6], being addressed with Inter-Operable Group References (IOGRs) in Fault Tolerant CORBA [16]. LIIs support the completability of invocations to other available and compatible invocation targets.

Group invocations, as used in this paper, refer to the ability to have a set of services receive the same invocation. Whereas group messaging has been used to distribute events, they often lack the desired distributed-object abstraction, cp. JavaGroups [7]. And while the concept of group invocations is common in parallel programming, e.g. Group Method Invocation [12], our motivation is not parallelism for performance *per se*, but rather the assurance that the entire group eventually receives the invocation, e.g. to change the configuration parameters or distribute data. Group invocations enhance completability while supporting the simplicity of object-orientation, e.g. in unreliable networking environments and in cases when the client cannot know which objects must receive the invocation.

Batch invocations support the grouping of multiple related invocations. With regard to completability, the entire group of related requests is viewed as a single entity, thus providing consistency for the invocation participants that either all or none are sent. This is advantageous, for example, in TII scenarios when connectivity cannot be guaranteed.

Partial or interim results may be desirable in such systems when long-duration invocations or large transfers are involved. For example, interim updates to the status of a request could be provided at critical transition points (e.g., “request being processed,” “request completed,” etc.). Partial results could include large amounts of

[1] In order to improve scalability, concurrency or asynchronous models can be used. However, concurrency models often lead to increased overhead such as thread management, context switching, dynamic memory allocation, and synchronization [20]. Asynchronous models - where the executing thread runs on a different CPU than the invocation thread, are preferable in scenarios where blocking may occur, such as service chaining.

chunked detector or measurement values. Partial results enhance completability (with regard to expected behavior) by supporting "best effort," while interim results provide the ability for clients to know what is occurring with their longer-duration request.

QoS provides the capability of specifying different quality levels for invocation delivery to support deterministic behavior, as in durability (e.g., surviving reboots), ordering (e.g., commit after set), prioritization (e.g., abort might have higher priority), reliability (e.g., retries) and deterministic semantics (e.g., exactly once). JMS [22] and CORBA AMI support similar types of properties.

While the use of messaging such as JMS instead of RMI would support the desired asynchronicity for scalability and remote communication completability, the use of messages increases programming complexity due to its lower abstraction level. The lack of an object-oriented invocation model results in implicit contracts, implicit coupling, and weak typing. This can make JMS-based systems more difficult to program and maintain correctly.

Hence, neither RMI nor JMS alone satisfies the desired properties in our context. However, the choice of a communication provider for an application in the scenarios we discussed is often critical and involves many factors. In order to further adoptability, the solution should leverage middleware capabilities already commonly available and support provider exchangeability.

Thus there exists a need for an invocation middleware that provides the simplicity and strong typing of an object-oriented paradigm while supporting the following properties:

- scalability of asynchronous invocations,
- Time-Independent Invocations (TIIs),
- Location-Independent Invocations (LIIs),
- group invocations,
- batch invocations,
- interim or partial results,
- QoS,
- client-service invocation model decoupling,
- communication provider exchangeability.

We have designed JILM as a solution to provide these properties. We will begin with a description of our general solution approach followed by a detailed solution. We will then evaluate JILM and compare it with other middleware and related work.

2 General Solution Approach

Our solution approach consists of the following participants (see Fig.1):

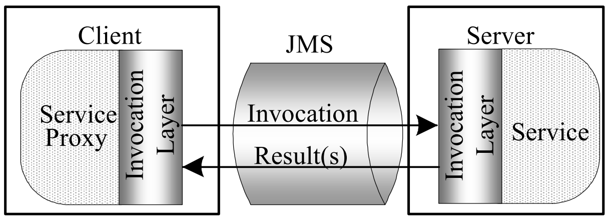

Fig. 1. General solution approach

JMS. JMS is used to transport the call, providing asynchronous communication and making the desired QoS properties available to services, including priority, filtering, call expiration time, call persistence, durability, etc. To support TII, the calls are queued by JMS until the intended peer becomes available to receive and process the calls. Similarly, queuing supports LIIs since any message consumer may process the call and the client is unaware of the service's true location or reference. Group invocations are supported by placing the calls in a topic (publish-subscribe), where multiple services retrieve the identical call. Batch invocations are supported by combining messages into a transacted session.

Invocation Layer. The object-oriented invocation layer addresses JMS deficiencies and supports decoupled client and server invocation models.

On the client, the method invocation is translated into a message that includes the service identifier, the method name, and parameters. On the server, the invocation layer retrieves the message and invokes the call on the appropriate service instance.

The following client invocation models are supported:

- *Synchronous*. Blocks the client thread until the response is received,
- *Asynchronous*. After the call is placed in a message, control is returned to the client thread while the response is received in a separate thread,
- *Asynchronous with multiple responses*. Multiple responses for the same invocation are returned in a separate thread to support interim or partial results,
- *Futures*. Creates a non-blocking call whereby the same client thread context is retained to retrieve the response. Via polling, the client can check if results are ready or block until ready.

The supported client invocation models are declared in the service proxy interfaces, which include the supported synchronous and or asynchronous method declarations. To be able to clearly distinguish a synchronous from an asynchronous invocation, separate interfaces can be used, and methods that provide an asynchronous invocation model must define a response handler as the first input parameter. Asynchronous responses are provided in the client when the invocation layer invokes the response handler.

The following server-side service invocation models are supported:

- *Synchronous*. Simplifies service programming and provides better performance for short duration calls, since it retains the thread context,
- *Asynchronous*. Enables pooling and reuse of threads that would otherwise wait, which can enhance server scalability in certain scenarios where service chaining or network calls are involved,
- *Asynchronous with multiple responses*. Multiple responses, such as partial results, may be returned before the final response.

The supported service invocation model(s) are defined by the service implementation, not the proxy. Thus a service may implement only a synchronous method, but may offer the client both synchronous and asynchronous method declarations. This decoupling of client and service invocation models can simplify the service implementation while providing flexibility for the client. Reflection, dynamic method signature mapping, and a preference strategy are used to decide which service implementation method corresponds to a client call.

Partial results are supported by associating multiple response messages with a single call context.

Service. The service provides functionality independent of the middleware used to invoke it. The service implementation defines the supported service invocation model for each method. During the registration of a service, the invocation layer uses reflection to dynamically determine the implemented method signatures in order to use the appropriate invocation model to invoke a method.

Proxy. The Proxy pattern [3] is used to represent the service on the clients. Since the mechanism for propagating a call is independent of the actual interface or method name, a java.lang.reflect.DynamicProxy is utilized, which supports different service interfaces with a common service proxy implementation of the client invocation layer described above. For each service, synchronous and/or asynchronous interfaces are provided (or a variant of one interface that provides both synchronous and asynchronous methods), allowing the client to choose the invocation model it wishes to use (per interface or per method).

Since the service configures the JMS-related QoS properties of the service proxy instance, by distributing the proxy, e.g. via Jini's Lookup Service [23] or JNDI [21], services can specify and control the QoS - which was not possible with pure JMS.

3 Detailed Solution

The design of JILM will be illustrated via client (Fig. 2) and server (Fig. 3) collaborations.

Client Invocation Layer Initialization. The client uses a lookup mechanism to retrieve a service proxy with the desired interface(s). The service proxy contains a Forwarder instance (ref. Forwarder-Receiver pattern [3]) that encapsulates a JMSConnectionFactory and the JMS properties to be used. It creates a connection and registers itself as a JMS message producer.

To receive the results of a method invocation, the Forwarder also creates a Receiver which creates a temporary destination for responses. A temporary destination exists as long as the JMS connection exists. The identifier of the response destination is sent via the JMSReplyTo header property of a JMS message.

For time-independent responses, the Receiver within the service proxy transparently makes a durable subscription to an alternative response topic. In order to receive only the client's response messages, a JMS message selector with a permanent client identifier is registered. This approach is also used to store responses when a JMS connection has been lost.

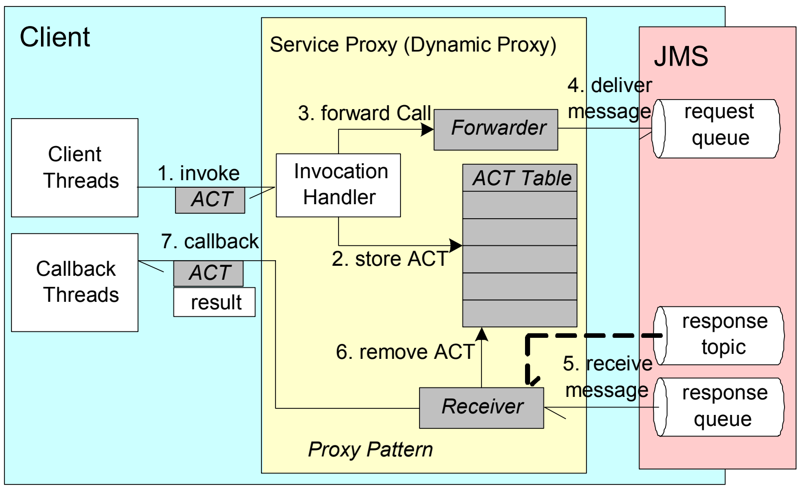

Fig. 2. Client invocation layer

Client-Side Invocation Collaborations. Fig. 2 illustrates the dynamic collaborations among participants in the client.

For an asynchronous invocation, the client explicitly creates and passes an Asynchronous Completion Token (ACT) [20], realized as a ResponseHandler, with an invocation (1). The Invocation Handler of the service proxy stores the ACT in the ACT Table (2) of outstanding requests and passes the call to the Forwarder (3). The Forwarder marshals the call arguments (service identifier, method identifier, and method arguments) and the ACT in a JMS message and delivers it asynchronously (4). The identifier of the response destination is sent via the JMSReplyTo header property of a JMS message, at which point control is returned to the client invoker.

With a synchronous invocation, the ACT is created internally by the service proxy and the thread is blocked until the response containing the desired ACT is returned.

When the response containing the ACT and the final result is returned (5), the Receiver demarshals the JMS message and removes the original ACT from the ACT Table (6). Then a callback thread notifies the client asynchronously about the result by utilizing the ResponseHandler's callback method (7).

To indicate interest in multiple responses, the client supplies a special ACT, realized as a MultiResponseHandler type, which is not removed until a message with a completion flag is received.

For futures, the client provides a special ACT, realized as a FutureResponseHandler type, where the result is stored until retrieved by the client thread.

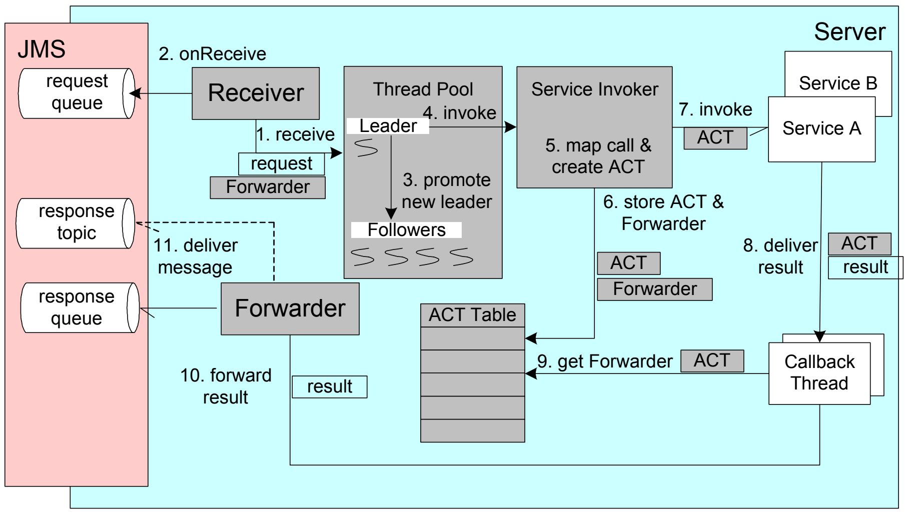

Fig. 3. Server invocation layer

Server Invocation Layer Initialization. A service registers with the invocation layer, which creates and assures distribution of a service proxy to clients. The method mapping table, which maps the service proxy interfaces onto the actual service implementation, is stored in an instance of a service invoker component. A Receiver is created that establishes a connection to the JMS server for a JMS message consumer at the pre-configured JMS message destination. It is responsible for demarshalling messages and maintains references to the service invokers to dispatch the incoming requests.

The Leader/Followers pattern [20] is utilized, where a Thread Pool is initialized and the leader thread calls the receive method of the Receiver (1), which itself invokes the synchronous receive method of the JMS message consumer (2).

Server-Side Invocation Collaborations. Fig. 3 illustrates the dynamic collaborations among participants in the server asynchronous invocation model.

When the JMS message arrives, the Receiver first demarshals the message. Since the message will be discarded, a Forwarder is created (if one does not already exist for this client) which contains the client JMS response destination. Since JMS does not support concurrent sends to one queue, only one Forwarder instance per client destination is created.

Based on the service identifier transmitted with the request message, the Receiver obtains the associated service invoker component. Then the invocation arguments (method identifier, call arguments, and the service identifier reference) along with the Forwarder are passed to the leader thread. This thread immediately promotes a new leader (3) and then becomes a processing thread invoking the service invoker (4). The service invoker creates a new ACT, realized as a ResponseHandler, which identifies the client response Forwarder (5) and is stored in the ACT table (6). The service invoker maps the call onto the actual implementation method and makes the invocation on the service (7). For an asynchronous method, a reference to the ResponseHandler is passed as the first parameter. Before method completion, an incomplete status can be set in the ResponseHandler, which will cause it to be retained in the ACT table for future retrieval, e.g. on completion of an asynchronous invocation to another service. The ResponseHandler can also be used to cause partial results to be sent to client. When the service has finished processing, the thread returns to the thread pool and becomes a follower thread.

A callback thread from a separate pool is used to send the result, thus decoupling service processing threads from communication issues. The callback thread retrieves the ResponseHandler from a queue, which contains the result along with the original client ACT of the invocation (8), and obtains the associated response Forwarder from the ACT table (9). Then it invokes the Forwarder's send method (10), which marshals and sends the result with the client ACT in a JMS message (11). Since JMS does not support concurrent message sending, the Forwarder's send mechanism is synchronized.

If the client destination is temporarily unreachable by the service, the Forwarder can be configured to either discard results or include an alternative JMS destination (such as a topic) to hold the responses until the client is available. The Forwarder must include a unique client identifier with the message to allow the client to select its responses.

4 Performance Evaluation

In this section we evaluate the performance and scalability of our JILM implementation, comparing the throughput of JILM to Java RMI two-way invocations and measuring the overhead of JILM versus a standalone JMS solution. For our measurements we utilized the Sun reference implementation for Java RMI, JDK 1.4.1_01 and the open source project OpenJMS v.0.7.5 [18] as a JMS implementation. Other JMS providers may perform differently.

JILM Overhead vs. JMS. We first determined the overhead of JILM itself by measuring the average throughput of 10,000 two-way invocations in a single-threaded client application using the asynchronous invocation model of JILM. For this experiment we utilized a 1.7 GHz computer with 768 GB of RAM running Windows XP.

The result was an average additional overhead of 900 microseconds per two-way invocation across both the client and service-side invocation layers versus the time used by JMS.

JILM vs. RMI (Two-Tier). To test the scalability of JILM clients, in this two-tier experiment we compared the throughput of 10,000 two-way invocations in a single-threaded client application using RMI, the asynchronous invocation model of JILM, and the synchronous invocation model of JILM. In order to simulate different call durations, the client invokes a simple method that takes a wait argument specifying a specific server delay before the method returns with a short sample string.

For this experiment we utilized a 1.53 GHz computer with 768 GB of RAM as client and a 1.7 GHz computer with 768 GB of RAM as server. Both were running Windows XP and were connected by a 10 Mbps Ethernet.

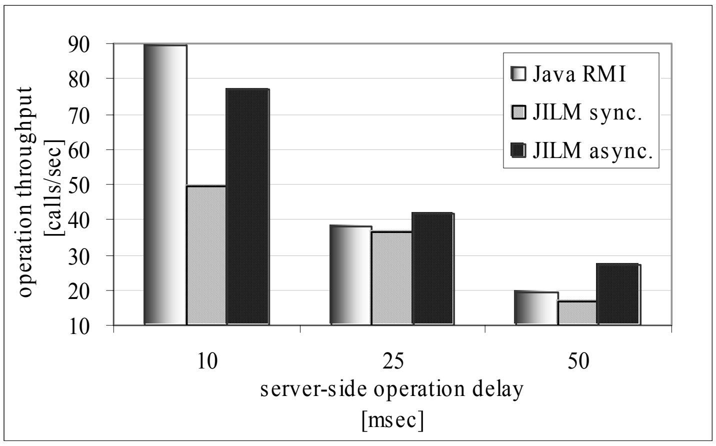

Fig. 4. Two-tier scalability results

Fig. 4 compares the empirical results. For short duration invocations, RMI has better performance than asynchronous JILM due to the overhead of the JILM+JMS implementations. This overhead is also apparent for synchronous JILM, whereby for longer duration calls it nears RMI's throughput.

Thus, once the server invocation delay exceeds the JILM+JMS overhead, the scalability advantages of asynchronous JILM become apparent. In addition, at those invocation delays synchronous JILM can be used to benefit from its simplicity and completability advantages with near RMI performance.

JILM vs. RMI (Three-Tier). In our next experiment, we show the effects of service chaining on scalability. We compared the throughput of 10,000 two-way invocations which were performed by a single-threaded client application using RMI and the asynchronous invocation model of JILM.

For this experiment the client application runs on a 1.53 GHz computer with 768 GB of RAM, the middle tier service on a 1.7 GHz computer with 768 GB of RAM and the sink server on a 1GHz computer with 1GB of RAM. During the measurements the server delay on the middle tier was set to zero and on the sink server it was set to 10, 25, and 50msec.

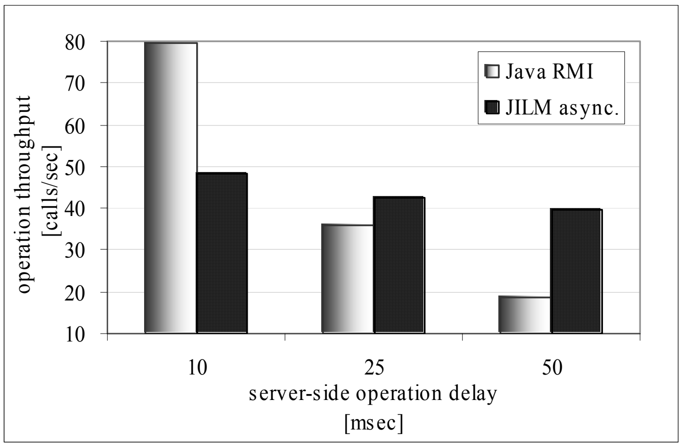

Fig. 5. Three-tier scalability results

Fig. 5 compares the empirical results. As in the first experiment, RMI has better performance for short duration invocations. However, as the server delay of the sink server increases, JILM is able to take advantage of asynchrony to achieve higher invocation throughput between the participants in comparison to RMI's blocked calls.

From our three-tier results we see that the use of asynchronous invocations is beneficial for throughput and scalability for cases where service chaining or similar effects occur.

5 Related Work

Table 1 shows a comparison of middleware based on our required set of features.

Table 1. Middleware comparison

Feature	JMS	RMI	Async RMI	CORBA AMI	JILM
Object-oriented invocations	N	Y	Y	Y	Y
Synchronous	Y	Y	Y	Y	Y
Asynchronous	Y	N	Y	Y	Y
Time-independent	Y	N	N	Y	Y
Location-independent	Y	N	N[2]	Y[3]	Y
Group communication	Y	N	N[2]	Y[4]	Y
Batch communication	Y	N	N	N	Y
Interim/partial results	Y	N	N	N	Y
QoS[5]	Y	N	N	Y	Y
Invocation model decoupling	N/A	N	N	Y[6]	Y
Provider exchangeability	Y	N	N	N[7]	Y

RMI. While callbacks have been used to support a form of asynchronicity within the constraints of the RMI implementation, a number of issues occur:

– clients must make an RMI server available (requires additional client resources),
– server threads may be blocked (e.g. on the callback for busy clients),
– client security issues (with ServerSockets) and firewall issues,
– correlating the client thread context with the response,
– client threads are blocked until the server responds to the initial JRMP call,
– concurrent calls utilize additional sockets.

So while callbacks reduce the client call blocking time equivalent to the server processing duration, as the system scales to capacity, these limitations will affect the system non-optimally.

RMI Variants. In contrast to callbacks, Futures [24], Batched Futures [2], and Promises [11] address call latency by delayed referencing of the result, thus retaining the client thread context from request to response.

Various asynchronous JRMP-compatible RMI variants have been created, such as ARMI [19], Active-RMI [8], and as described by Kerry Falkner [10]. They rely on a modified RMI stub compiler to insert asynchronous behavior and create a separate thread for each invocation. E.g., whereas [10] is an asynchronous RMI

[2] Possibly supported by parallel RMI extensions.
[3] With the addition of FT CORBA [16].
[4] With the use of Data Parallel CORBA [17].
[5] E.g., ordering, prioritization, delivery guarantees, durability, rerouteability, etc.
[6] Not included in the specification, but demonstrated in [4].
[7] Implementations can be exchanged, but the protocol is fixed.

implementation that uses Futures, JILM supports Futures without adding Java keywords to the language and without creating a new thread per outstanding request.

JR [9] extends the language to provide a concurrency model on top of RMI, but still has the underlying RMI limitations (although JR is capable of using another transport).

Custom-protocol variants that support asynchronous communication include NinjaRMI [15], which requires language extensions. It uses one-way asynchronous communication that may include callback information.

Although RMI does not support group method invocation, much work has been done to create extensions to RMI to support parallel programming environments, e.g. Manta [14], GMI [12], and CCJ [13]. However, our intent and usage model is not parallelism *per se*, and these specialized extensions were unsuitable for use in our context, not fully supporting model decoupling, QoS, and provider exchangeability.

CORBA AMI. While CORBA AMI provides asynchronicity, it requires an IOR or IOGR in order to make an invocation [6], limiting its LII capability. On the other hand, JILM does not require a valid service reference, but can store the invocation in JMS until a service instance exists and retrieves it. Neither batch invocations nor partial or interim results are supported.

While CORBA Messaging implementations could be exchanged, the middleware protocol is specified, whereas JMS does not specify the protocols used, thus JILM provides a greater degree of provider flexibility. For Java environments, Java ORB implementations that fully support the AMI specification are not currently known to us or in wide use at this time.

Deshpande [4] describes an asynchronous method handling (AMH) mechanism for decoupling client invocations from server invocations, in particular for middle-tier servers. JILM's service invocation layer supports an equivalent capability for Java services while supporting interchangeability of JMS providers or other middleware via the Forwarder-Receivers.

6 Conclusion

RMI has not addressed the demands in today's systems for asynchronicity and remote invocation completability. In particular, it lacks the desired properties as shown in Table 1.

While JMS has been used in distributed computing to address some of these issues, its usage entails issues including the lack of an object-oriented invocation model, implicit contracts and coupling, and weak typing. This is the area that JILM addresses. By providing an invocation layer around JMS, we were able to mitigate many issues related to JMS, while supporting the desired (a)synchronous invocation models and completability properties.

Our performance measurements show that JILM adds 900 microseconds in roundtrip overhead to the JMS implementation. The asynchronous JILM scales well as the call duration increases. Our three-tier results showed that an asynchronous invocation model is beneficial for throughput and scalability for cases where service chaining or similar effects occur. In addition, the JILM synchronous model can be used for simplicity and yet achieve near RMI throughput for longer server call durations while benefiting from JILM's completability advantages.

By relying on patterns for JILM, our design can be readily applied and reused on various platforms. For systems or architectures that are considering JMS usage, JILM provides an easier and higher-level programming model than direct messaging. In addition, the use of object-oriented invocations ensures that the client utilizes the types and methods expected by the service and allows the service to provide a client proxy that encapsulates the communication mechanism expected by the service. One-way calls could easily be supported if desired.

JILM addresses both the need for asynchronous invocations in Java and the need for remote invocation completability assurances in today's systems.

References

1. Arulanthu, A. B., O'Ryan, C., Schmidt, D.C., Kircher, M., Parsons, J.: The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging. In Proceedings of the IFIP/ACM Middleware 2000 Conference (2000)
2. Bogle, P., Liskov, B.: Reducing Cross Domain Call Overhead Using Batched Futures. In Proc. OOPSLA'94, ACM SIGPLAN Notices, volume 29 (1994)
3. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture - A System of Patterns, Wiley and Sons Ltd. (1996)
4. Deshpande, M., Schmidt, D.C., O'Ryan, C., and Brunsch, D.: The Design and Performance of Asynchronous Method Handling for CORBA. In Proceedings of the Distributed Objects and Applications (DOA) conference (2002)
5. Eckerson, W.W.: Three Tier Client/Server Architecture: Achieving Scalability, Performance and Efficiency in Client Server Applications. In Open Information Systems, vol. 10 (1995)
6. Gore, P., Cytron, R., Schmidt, D., O'Ryan, C.: Designing and Optimizing a Scalable CORBA Notification Service. In Proceedings of the ACM SIGPLAN workshop on languages, compilers and tools for embedded systems (2001) 196–204
7. JavaGroups Web Site: http://www.javagroups.com
8. Karaorman, M., Bruno, J.: Active-RMI: Active Remote Method Invocation System for Distributed Computing using Active Java Objects. In TOOLS USA (1998)
9. Keen, A., Ge, T., Maris, J., Olsson, R.: JR: Flexible Distributed Programming in an Extended Java. In Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (2001)
10. Kerry Falkner, K.E., Coddington, P.D., Oudshoorn, M.J.: Implementing Asynchronous Remote Method Invocation in Java. University of Adelaide (1999)
11. Liskov, B., Shrira, L.: Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In Proc. SIGPLAN'88 Conf. Programming Language Design and Implementation (1988) 260–267
12. Maassen, J., Kielmann, T., Bal, H.E.: GMI: Flexible and Efficient Group Method Invocation for Parallel Programming. In LCR-02: Sixth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers. Lecture Notes in Computer Science, Springer-Verlag, Berlin Heidelberg New York (2002)
13. Nelisse, A., Maassen, J., Kielmann, T., Bal, H.E.: CCJ: Object-based Message Passing and Collective Communication in Java. In Concurrency and Computation: Practice and Experience, Vol. 15, Issue 3-5 (2003) 341–369

14. van Nieuwpoort, R., Maassen, J., Bal, H., Kielmann, T., Veldema, R.: Wide-area parallel computing in Java. In Proc. ACM 1999 Java Grande Conference (1999) 8–14
15. NinjaRMI Web Site: http://www.eecs.harvard.edu/~mdw/proj/old/ninja/index.html
16. Object Management Group: Common Object Request Broker Architecture (CORBA) Core Specification, 3.0 ed.
17. Object Management Group: Data Parallel CORBA Specification. (May 1, 2002)
18. OpenJMS Web Site http://openjms.sourceforge.net/
19. Raje, R., Williams, J., Boyles, M.: An Asynchronous Remote Method Invocation (ARMI) Mechanism for Java. In Proc. of the ACM 1997 Workshop on Java for Science and Engineering Computation (1997)
20. Schmidt, D., Stal, M., Rohnert, H., Buschmann, F.: Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects. Vol. 2. Wiley and Sons Ltd. (2000)
21. Sun Microsystems: Java 2 Platform Standard Edition version 1.4.1
22. Sun Microsystems: Java Message Service Specification, version 1.1 (2002)
23. Sun Microsystems: Jini Architecture Specification, version 1.2 (2001)
24. Walker, E. F., Floyd, R., Neves, P.: Asynchronous Remote Operation Execution In Distributed Systems. In Proc. of the Tenth International Conference on Distributed Computing Systems (1990)