Voyager Architecture Best Practices

By Thomas Wheeler

Recursion Software, Inc.
March 30, 2005

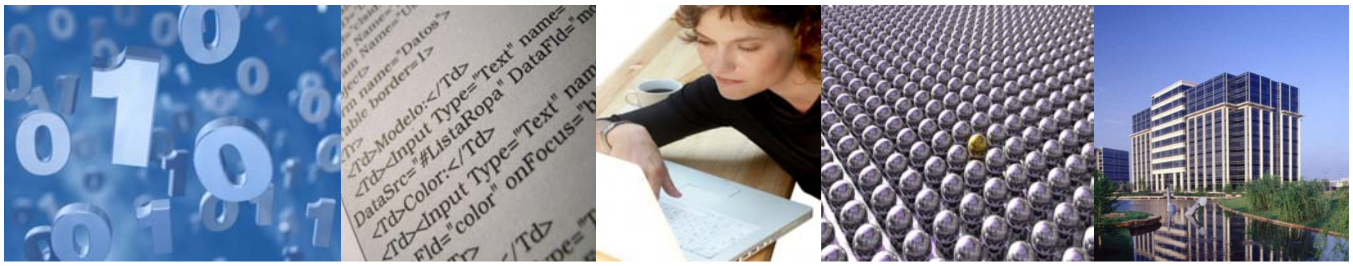

TABLE OF CONTENTS

Voyager Architecture Best Practices

By Thomas Wheeler

Introduction

Java is the language and environment of choice today for developing enterprise software systems. *Voyager* is a Java-centric platform for developing distributed software systems. This paper describes some architectural best practices when developing software systems with *Voyager*. These best practices are described in the context of *Voyager* features and capabilities.

Messaging

Voyager provides support for several messaging types. Synchronous messaging consists of a single invocation that blocks until the response is received from the server. Asynchronous messaging consists of a single invocation that does not block. Publish/subscribe messaging consists of publishing (sending) messages to one or more subscribers. In addition to these messaging types, *Voyager* provides support for object mobility.

Each of these messaging types has different implications for application architecture. During architectural analysis, carefully consider the responsibilities for each component, how these components will need to communicate, and the appropriate messaging to use for communication between components. The following guidelines can be applied:

- Assignment of responsibilities impacts the associations and relationships between components, which in turn affect network usage.
- Use synchronous messaging for transactional operations, and asynchronous messaging for non-transactional or long-lived operations.
- Use coarse-grained remote interfaces for synchronous messaging.
- Use publish/subscribe to send notifications or information when there are multiple interested parties and/or the publisher may not know the interested parties.
- Use object mobility and agents to distribute processing among multiple systems or when an operation requires participation by several systems

Using coarse-grained remote interfaces for synchronous messaging reduces network "chatter", increasing perceived application performance while reducing usage of valuable network bandwidth. When defining interfaces, consider the need for remote communication. Careful assignment of responsibility and associations between components to limit remote messaging will result in increased performance and scalability. Using well-defined coarse-grained interfaces also results in an architecture with reduced coupling between components and increased internal component cohesion. Choosing appropriate messaging strategies will increase application performance, scalability, and robustness.

Synchronous messaging forces a client to wait for the server to finish processing a request before continuing, and depends on the availability of the server and a reliable network connection. Consequently, synchronous messaging is best used where it is essential to verify that the request was successfully processed. Asynchronous messaging, or "fire-and-forget" messaging, allows the client to continue processing: it does not wait for a response from the server. Asynchronous messaging is a good choice for two scenarios:

- No response is required from the server. This is appropriate for notification messages, informational messages, or other scenarios where a failure to receive or process the request does not impact primary functionality.
- Long-lived transactional operations. Synchronous messages lock a connection for the time it takes to send the request, process it, and receive the response, but the connection is not in use while the request is being processed and could potentially be used for handling other messages. Using asynchronous messaging for long-lived messages frees up the connection (increasing scalability); reduces the impact of network outages (increasing robustness); and potentially increases perceived application performance. The response can be obtained by polling, publish/subscribe, or callback.

Voyager's publish/subscribe messaging supports efficient broadcast of messages (events) to a number of subscribers contained in a virtual container called a *Space*. A *Space* consists of two or more connected *Subspace*s, with each *Subspace* usually being in a different process. Messages published to any *Subspace* are published to all members of the *Space* and to all objects within each *Subspace*. This mechanism is scalable, high-performance, and flexible. Publish/subscribe messaging is appropriate for broadcasting application events to a number of receivers that may or may not be known by the publisher. Publishing may be one-to-many (for example, a server broadcasting to multiple clients) or many-to-many (multiple processes broadcasting to multiple receivers, such as a peer-to-peer system). The two basic *Space* topologies are star (and binary-star) and ring (double-ring). These basic topologies can be used to form large, complex topologies that are scalable and reliable.

A *mobile object* is capable of moving (or being moved) from one process to another. An object consists of both code and data. In *Voyager*, mobile objects carry both their state and functionality (data and code) when moving to a remote process. This allows constant, dynamic updates to application functionality. Mobility also enables the creation of *mobile agents* that autonomously move among a set of processes. In a mobile agent system, the primary actors, or controllers, are the agents themselves as opposed to traditional systems where control is typically assigned to a centralized component. Architecturally, this allows the creation of extremely dynamic, decentralized, and flexible systems. Consider using mobile agents in situations where these attributes are desirable or necessary.

Consider an application where a server component obtains data from a database on behalf of a client. The client user interface presents the data to the user and accepts input from the user. Operations are performed on the data, and the server then updates the database. In this scenario, the key decision is whether the server or client is to be responsible for performing operations on the data – or whether they will share in this responsibility. This decision affects how much data is transferred between the server and clients, overall application architecture, performance and scalability, security concerns, and more. Operating under the assumption that the client will be responsible for these operations, the following practices should be considered:

- Pass large chunks of data between the server and client instead of small pieces. Using a remote call to obtain the value of a single field is highly inefficient; it is better to pass whole objects (e.g. records, or rows) or possibly even the entire result of a query.
- Use synchronous messaging to ensure updated data is stored. Failures (exceptions) can be propagated back to the client, which can then take appropriate action (retry; notify the user of the failure; use temporary storage in the filesystem, etc.)
- Use asynchronous messaging to send queries to the server. Results can be obtained by polling the server or receiving notification from the server.
- Use publish/subscribe messaging to notify other clients of changes to data. The server can act as the central hub of a star topology.

- Use mobile objects to allow the server to create objects, which will act on the data and move those objects to the client. The server then becomes responsible for choosing operations, and the client is responsible for performing them. This pattern is useful when business processes change frequently, as it limits new deployments to the server.

Remote References

Voyager provides several ways to acquire a remote reference. A remote reference is represented in *Voyager* by a *Proxy* object. There are three primary ways to acquire a remote reference:

- Naming service
- Factory service
- Perform a remote invocation

Consider the object's lifecycle and responsibility for creation of objects when determining how to acquire a remote reference. In general, the following guidelines apply:

- Use *Voyager*'s Naming Service for binding server-side objects to a well-known name, making them available to any (or all) clients. Objects bound into the Naming Service should be long-lived and stateless.
- Use *Voyager*'s Factory Service when the client should have responsibility for creating an object. (The client knows when and/or how to create the object.) The object can be created at a determined location, or at the location of another remote object.
- A remote reference can be acquired as the return value of a remote invocation. In this scenario, the client calls a remote object that returns an existing or new object. Use this mechanism when the object should only be made available to a specific client or when its lifecycle is based on client need. The server has responsibility for creation of the object.

In addition to acquiring a remote reference, the architecture should also take into consideration the number of remote references per client. A remote reference uses resources (primarily memory); unbounded creation of remote references can have a significant impact on application scalability and performance. *Voyager* imposes no limits on the number of remote objects that can be created, but each remote object will consume system resources – especially memory. Excessive numbers of remote objects also implies excessive remote messaging. Consider the following techniques to reduce remote object creation:

- Use stateless objects bound into the *Voyager* Naming Service. Multiple clients can simultaneously utilize a stateless object.
- Use coarse-grained interfaces to aggregate functionality.
- Consider passing state information from client to server rather than creating stateful server objects. (However, balance this against the impact on network utilization and performance.)
- Consider using publish/subscribe instead of calling remote objects directly.

Messaging Protocols

Voyager provides support for several messaging protocols: SOAP, JRMP, IIOP, and VRMP. JRMP and IIOP are used when interfacing with external RMI or CORBA systems. SOAP is the standard protocol for Web Services communications. Use SOAP when Web Services support is required (either accessing remote Web Services or servicing Web Services clients). VRMP is *Voyager*'s native binary messaging protocol, and is the default protocol for *Voyager*-to-*Voyager* communications.

Resource Utilization

Modern enterprise applications are often required to scale to hundreds or thousands of users. Scalability is constrained by the resources of the deployment environment. This includes memory, processing power, network bandwidth, sockets, and threads. To maximize scalability, consider the following practices:

- Manage socket and thread utilization through configuration of *Voyager*'s connection management policies.
- Manage memory utilization by controlling per-client object creation and tuning distributed garbage collection.
- Use appropriate messaging strategies for remote communication.

Voyager provides connection management facilities that allow policy-based control of network connections (sockets). A connection management policy supports the following parameters:

- *MaxClient* – the maximum number of client connections to remote processes.
- *MaxServer* – the maximum number of server connections from client processes.
- *MaxIdle* – the maximum number of idle client connections.
- *IdleTime* – the maximum time a connection is allowed to be idle before being closed.
- *ServerIdleTime* – the maximum time a server connection is allowed to be idle before being closed.

The following policy settings support increased scalability:

- Minimize the number of simultaneous client connections by setting *MaxClient* to a low value. *Voyager* will automatically create new network connections on an as-needed basis for multi-threaded clients. By limiting the number of simultaneous connections, the server will be able to support more clients simultaneously.
- Reduce the idle time for client connections by setting a low value for *IdleTime*. Closing unused connections quickly will allow the server to reclaim resources and make them available for additional users.

For more details on connection management, refer to the `com.objectspace.voyager.tcp` package in the *Voyager* API documentation.

Voyager automatically creates proxy objects which handle messaging between the client and server. For each server-side object referenced by a client, *Voyager* creates a proxy object. When architecting a distributed application using *Voyager*, carefully consider which server-side objects will be accessed remotely. Minimizing the number of objects accessed remotely will reduce memory requirements on the application.

Voyager provides automatic distributed garbage collection facilities. Server objects are automatically released when they are no longer referenced by any clients. There is a configurable delay (defaulting to two minutes) between the time when the object is no longer referenced and the time *Voyager* releases the object. By reducing this delay, objects will be reclaimed more rapidly. Consider this option if the server frequently creates short-lived remote objects. (But first consider ways to reduce the number of objects created.) For details, see the `com.objectspace.voyager.vrmp.dgc` package in the *Voyager* API documentation.

Security

Security in a distributed application covers several areas. Encryption is essential for protecting valuable information as it is transmitted across the network. Authentication is essential for validating client identity. Authorization is essential for preventing inappropriate usage of services or information access. *Voyager Security* provides capabilities in all these areas.

Voyager Security provides support for SSL socket factories; tunneling through SOCKS4, SOCKS5, and HTTP; and a framework and policy for authentication and authorization.

SSL or tunneling should be used for any *Voyager* server process open to external client access. (External is a relative term: it may mean outside a department, outside a site, or outside an enterprise.) This prevents unauthorized and unauthenticated access. Depending on the sensitivity of the information being transmitted, SSL should be used for encryption.

Summary

Architecting a distributed application requires careful consideration of performance, scalability, reliability, security, maintainability, inter-operability, and extensibility. Making appropriate choices in tools and technologies – and their use – is important. *Voyager* provides the flexibility and features for a wide variety of needs. This "best practices" paper provides the background necessary to make appropriate choices when using *Voyager* to develop a distributed application. Specifically covered are considerations for messaging types, remote references, messaging protocols, resource utilization, and security.

About the author

Thomas Wheeler is a senior software engineer at Recursion Software, Inc. He may be contacted by email at engineer@recursionsw.com.

Recursion Software, Inc.
2591 North Dallas Parkway
Suite 200
Frisco, Texas 75034
1.800.727.8674 or 972.731.8800
www.recursionsw.com