Selecting Between Synchronous and Asynchronous Alternatives

Peter Houston, Microsoft Corporation

Introduction

With the trend towards distributed computing, and the emergence of powerful application-to-application communication technologies (e.g., message queuing middleware and object request brokers), companies have to make complex architectural design decisions. One important decision is selecting between synchronous and asynchronous forms of communication. Developers observe the proliferation of asynchronous message queuing middleware products in the last few years from IBM and Microsoft. However, at the same time, developers wonder when these asynchronous strategies are *not* appropriate. This question is very difficult to answer intuitively and requires deeper examination.

Why Is This a Hard Problem?

To understand why this is a hard problem, consider just two forms of communication: synchronous remote procedure calls (RPC) and asynchronous message queuing middleware (MQM) technologies. They are representative of the majority of communication technologies on the market and are nearly *opposites* of each other in their relevant characteristics. RPCs are natively synchronous and cause the calling application to block until a response comes back from the receiver. MQM-based requests are natively asynchronous and do not block the sender of the request in any way. However, design decisions become obscured because it is possible for the RPC to appear non-blocking and asynchronous to the original thread/process. Also, an MQM message may appear blocking when its calling application waits for a response. Furthermore, higher-level abstractions (e.g. COM or Java components) can sometimes hide lower-level methods (e.g. RPC or MQM) used for communication. This makes design decisions even more obscure.

Connection-Oriented Versus Connection-Less Communication

To make good decisions about communication technologies, designers need to look deeper and focus on several important issues. The first is to understand the difference between *connection-oriented* and *connection-less* communication technologies.

Connection-oriented technologies require requestors to locate the receiver on the network, bind to an instance of the receiver, and share state for the duration of the request. State is shared because the requestor and receiver use shared information (i.e. input and output parameters) to make requests and responses, and both requestor and receiver can act together on that information.

Connection-less technologies do not require requestors to locate or bind to specific instances of receiver applications in order to make a request. Receivers also don?t need to be reachable via an active network connection, or running, at the time a request is made - which means that connection-less communication is inherently asynchronous and one-way (i.e. results cannot be returned via the same request). Connection-less technology precludes any shared state between requestor and receivers. All state must be encapsulated in the message itself because no contact back to the requestor will occur when the receiver processes the request. In order to be useful, connection-less technologies should also provide some form of recoverable, store-and-forward capability to ensure that requests do not get lost before the receiver can be contacted.

Because RPCs expose blocking, synchronous semantics to developers, and require requestor-receiver binding operations prior to communication, most RPCs are connection-oriented. And, because MQM implementations expose non-blocking, asynchronous semantics, and only require that target queue names be specified, they operate in a connection-less fashion.

Applications can be written to make RPCs appear non-blocking to the requesting process/thread, and to make MQM-based requests look synchronous, but when you stretch the technologies in this way, the results are not always so beneficial.. The strengths and weaknesses of each form of communication are shown in Table 1.

Technology	Strengths	Weaknesses
Blocking, synchronous based on connection-oriented technologies (e.g., RPC alone)	• Immediate confirmation of request outcome • Simplified requestor error handling • RPC-related benefits (e.g. strongly-typed interfaces)	·A network connection to receiver instance must be available ·Receiver instance must be running ·Requesting process/thread must wait for responses
Non-blocking, asynchronous based on connection-less technologies **(e.g., MQM alone)**	• Requests do not need to be targeted at specific receiver instances • Network connections do not need to be available at time of sending • Receivers don?t need to be running for senders to make requests • MQM-related benefits (e.g. heterogeneous platform support)	·The time when requests are processed is unpredictable ·Receivers cannot depend on timely responses ·Requester must depend on receivers to eventually process requests
Non-blocking, asynchronous based on connection-oriented technologies (e.g., RPC & sub-processes or threads)	• Requests appear non-blocking and asynchronous to the main requesting process/thread • RPC-related benefits (e.g. strongly-typed interfaces)	·A connection must be made to a specific instance of a receiver ·If requestor shuts down or fails before connection is made, request is lost
Blocking, synchronous based on connection-less technologies (e.g., MQM and application logic that waits for response messages)	• Requests appear blocking and synchronous to the main process/thread • MQM-related benefits (e.g. heterogeneous platform support)	·Error handling can be very complex ·Receivers need to be reachable and running ·Apps must develop private request/response protocol/conventions.

Table 1: Connection-Oriented versus Connection-Less Communication

From the table, two conclusions can be drawn:

- *Connection-oriented communication technologies appear to work best when receivers are expected to be reachable and running under normal conditions and when responses are required by requestors before they can continue processing.*
- *Connection-less communication technologies appear to work best when receivers are not expected to be available and when requestors do not need immediate responses (or any response) back from receivers.*

•

It is also important to note that with both connection-oriented and connection-less technologies, if responses are needed by requestors before they can continue processing, physical infrastructures must be designed to ensure that receiving applications remain reachable and running.

Transaction Processing Alternatives

It is tempting to read the preceding conclusions and believe communication decisions can be made based on connectivity issues alone. Unfortunately, this is not the case. The second area that needs examination is the type of transaction processing architecture required by the application. The different types of transaction processing impose restrictions on the type of communication technology that can be used. From a transaction processing perspective, most distributed application architectures fall into one of the following four groups.

No Transactions

With this style of application, neither requestors nor receivers use transactions of any form. If a resource such as a database fails to update properly, or a communication failure occurs, the application is simply out of luck.

Remote Transactions

In this case, the requesting application does not use transactions, while the receiving application does use transactions. An example would be an application that uses 3270 terminal emulation software to call a CICS mainframe application. The transaction on the mainframe will commit or abort completely autonomously from the requesting application.

Distributed Transactions

With this style, all work performed by both the requestor and the receiver is part of a single transaction. If any resource accessed by either the requestor or the receiver is unable to commit its changes, or there is an error such as a communication failure, or a timeout value is exceeded, the entire transaction fails and all resources must roll back to their prior state.

Distributed transactions are always synchronous. The requestor and the receiver must be in synchronous contact with each other and commit or abort as a unit. If the receiver is unreachable or not running, by definition, the transaction must fail. Examples of distributed transactions include applications written to use RPCs that support transactions (e.g., Encina Transactional RPC), *synclevel 2* extensions of the LU6.2 protocol (e.g., CICS Distributed Program Link), or other transactional protocols (e.g., Tuxedo XATMI or Microsoft?s DCOM when used with Microsoft Distributed Transaction Coordinator).

Transactional Queuing

With transactional queuing, applications use asynchronous message queuing technologies to communicate - with one critical requirement. All operations that send messages to, or receive messages from, a queue must be contained in a transaction. (Furthermore, the message queuing technology must support transactional operations).

If the requestor?s transaction commits, then all changes to resources such as databases are made permanent and the message (or messages) is released for delivery. If the requestor?s transaction aborts, all resource updates are rolled back and all messages sent within the transaction are discarded (and never delivered). In a similar way, if the receiver?s transaction is successful, all resource updates are made permanent and any messages retrieved by the receiver are considered as delivered. If the receiver?s transaction aborts, all resource updates are rolled back and all messages retrieved by the receiver are placed back on the queue (and considered as undelivered).

Transactions that abort within the receiver have absolutely no effect on the requestor - the requestor?s transaction would have to complete fully in order for the message to be sent to the receiver at all. Aborts within the receiver only cause messages to be returned to the queue for subsequent re-retrieval by the receiver.

Transactional queuing features can be found in products such as Microsoft?s Message Queue Server (MSMQ) and IBM?s MQSeries.

What Doesn?t Work

It is important to note that distributed transactions *cannot* be performed using transactional message queuing for communication between requestor and receiver. If a requestor begins a transaction, sends a message, waits for a return message, and then ends the transaction, nothing will happen. The requestor?s original message will not be released until the transaction commits - which will never happen because the requestor will still be waiting for a response to the message that hasn?t been (and will never be) sent. If the requestor wants to continue processing while waiting for the distributed transaction to commit or abort, it must use techniques such as sub-processes or threads and deal with the downsides highlighted above.

In a similar way, transactional queuing makes little sense when performed via connection-oriented technologies. Connection-oriented technologies require simultaneous connections between requestor and receiver - eliminating the benefits of asynchronous technologies such as message queuing middleware. If there is a chance that a network connection will be unavailable, or the receiver may not be running, then connection-less communication technology is effectively required.

The relative strengths and weaknesses of the four transaction processing models that do work and make sense are shown in Table 2.

Type of Transaction	Strengths	Weaknesses
No Transactions	• No management of resource updates is required • There are no data locking issues • Very simple to program	·There is a significant risk that some updates will succeed and others will fail resulting in inconsistent data. ·No data locking protection ·It may be hard for the requestor to know when something has gone wrong in the receiver (e.g. outcome is uncertain unless response communications containing results come back)
Remote Transactions	• Data integrity is preserved in the receiver • Requestor logic remains simple • Requires minimal hardware resources for the requestor	·It may be hard for the requestor to know when something has gone wrong in the receiver ·It is not possible to synchronize data updates when made by both the requestor and the receiver
Distributed Transactions	• Data integrity is preserved across both requestor and receiver • Simplifies error detection and correction	·Can require additional requestor hardware resources ·Creates many dependencies on networks and applications being available ·Locking conflicts can occur and are hard to fix ·Programming complexity in heterogeneous environments ·Can lock distributed system resources for an arbitrary length of time (as long as it takes for transaction to succeed/fail over unreliable net)
Transactional Queuing	• Data integrity is preserved locally in both requestor and receiver • Greatly reduced dependency on networks and applications being available • Greatly reduced chance of locking conflicts	·Synchronization of updates between requestor and receiver cannot be guaranteed ·In doubt states can arise where the requestor is unsure of the fate of its messages ·Reconciliation logic is required to restore global system integrity

Table 2: Transaction Processing Model Comparison

Are We Done Yet?

In a word, *no*. To this point, the article has focused on identifying technologies and architectures but has not made recommendations. Unfortunately, in order to make recommendations, a final issue must be identified and discussed. This issue is the type of the response required by the requestor from the receiver.

There are three possible levels of response required by requestors:

- **Immediate Response Needed** - A response is required from the receiver before the requestor can continue on to perform other work. An example would be a branch banking system that needs account balances to be retrieved from a centralized database before allowing any check cashing operations.
- **Eventual Response Needed** - A response is needed at some time in the future. The maximum amount of time that can be allowed to elapse is usually defined by a business policy rather than a technical requirement. For example, a company may dictate that salespeople must get order confirmations from their own warehouses within 24 hours of placing an order.
- **No Response Needed** - Assuming that the request will eventually reach the receiver, no response is needed. The requestor should be allowed to continue regardless of network availability or the receiver being active and trust that request delivery will occur. For example, an application that processes change-of-address notifications may need to notify a billing application to send invoices to a new address -- but has no interest in whether the billing application acts appropriately when it receives the notification.

Recommendations

In order to identify a communication technology, an application designer should *first* determine the type of transaction processing architecture required, then determine the level of response required by receivers, and finally review the plusses and minuses of connection-oriented and connection-less technologies.

Assume for simplicity that:

- *RPC* denotes technologies that natively exhibit blocking, synchronous and connection-oriented behavior (including ORB-based remote method invocation (RMI) capabilities and today?s DCOM from Microsoft) and
- *MQM* denotes technologies that natively exhibit non-blocking, asynchronous and connection-less behavior

Table 3 maps transaction processing architectures against response requirements and recommends a communication technology.

Type of Transaction	Immediate Response Needed	Response Needed Eventually	No Response Needed
No Transactions	RPC	MQM, 2-way messaging	MQM, 1-way messaging
Remote Transactions	RPC	MQM, 2-way Messaging	MQM, 1-way messaging
Distributed Transactions	Transactional RPC (or equivalent)	Transactional RPC (or equivalent) combined with sub-processes or threads	Transactional RPC (or equivalent) combined with sub-processes or threads
Transactional Queuing	MQM, 2-way messaging, transactional send & receive, (and highly available networks and applications)	MQM, 2-way messaging, transactional send & receive	MQM, 1-way messaging, transactional send & receive

Table 3: Communication Technology Recommendation

Two Caveats

There are two caveats to the conclusions made in the table above. First, when distributed transactions are combined with eventual-response and no-response models, there are two significant downsides:

- The requestor needs to stay running until the sub-process or thread can contact the receiver and commit all transactions.
- There may be a significant risk of deadlocks on the requestor since data is locked until the entire transaction commits.

Second, immediate-response-required scenarios that involve spanning heterogeneous platforms, application architectures and political boundaries may be addressed most efficiently with 2-way MQM-based approaches. There is a definite overhead to doing so -- mostly related to the additional error detection and correction code required to deal with *in-doubt* states -- but this can be a small price to pay given the comparative ease with which MQM-based technologies bridge between dissimilar entities (including people!)

The Bottom Line

After all of the analysis, most of what is said above can be boiled down into a few simple rules of thumb:

- When doing analysis, remember that even simple applications may require more than one type of communication technology; each call between a requestor and a receiver must be evaluated separately from the others in the application.

- If a requestor needs a response before it can move on to other work, connection-oriented technologies typically work best. That said, designers must make sure that networks and receiver applications will be *highly* available.
- If networks and receiver application will not or cannot be highly available, or the need for immediate responses can be avoided, connection-less technologies are probably the best match. In fact, finding ways to proactively reduce the need for immediate responses often translates into availability and performance gains for the requesting application. If the receiver does not have to be reachable and running, the requestor can function much more efficiently.

Finally, be careful about designing applications that require distributed transactions. The more application components, machines and network links that are involved, the greater the chance that at least one will fail. Because transactions are all-or-nothing in nature, even a single failure will require every component in the transaction to abort. In this sense, distributed transactions create the most rigid demands of all of the architectural models. If they are required, minimize the number of participants in each transaction through the use of options such as transactional message queuing.