Interop Testing Specification

Qpid Interop Testing Spec. Working Copy.

Draft.	Rupert Smith.	22nd Feb 2007	Document started.
Working Copy.	Rupert Smith.	6th Mar 2007	Document updated from feedback to draft on qpid-dev list. Last requirement # used: 47
Working Copy.	Rupert Smith.	7th Mar 2007	Senders and receivers to send reports to coordinator. Reply-to added to broadcast messages. Last requirement # used: 49
Working Copy.	Rupert Smith.	13th Mar 2007	Added test case names. Last requirement # used: 52
Working Copy.	Rupert Smith.	25th Sept 2007	Added test cases for message size variation. Last requirement # used: 60
Version 2 Work in Progress.	Rupert Smith	22nd Nov 2007	Test framework being expanded to cover functional and perforance tests and a much wider variety of testing possibilities.

Introduction:

The requirements in this specification use a common format, an example of which is given below:

RE-1.	**Sample Requirement.**	A brief descritpion of the requirement.

The requirements are numbered from 1.

Purpose:

- Test sending from and receiving by each of the clients in Qpid over both of the broker implementations.
- Enable testing of any JMS compliant product, by keeping a pure JMS sub-set of the testing framework seperate. This only applies to Java messaging client implementations.
- Provide a parameter driven test framework, that can be used to generate many testing scenarios for different messaging modes.
- Allow functional testing of messaging at the product surface, through the standard interfaces/protocols (JMS, AMQP), so that the same test suite may be applied over different implementations.
- Allow tests to be posed in terms of abstract asynchronous messaging concepts that AMQP and JMS support, rather than at the level of direct interfaces. This allows the same tests to carry forward as standards and products evolve.
- Enable interopability testing between any AMQP compliant components, not just those in Qpid.
- Allow performance testing to be carried out accross a distributed set of edge nodes connected to a messaging broker.
- Make tests robust enough to run as part of an automated build. The scripts should pass or fail, not hang, wait forever, run out of memory or otherwise cause an automated build process to fail to complete.
- Be capable of running the full test suite on several machines in a hands free way. In particular C++ tests need to run on unix and .Net on windows, necessitating a multi-box solution for full interop testing.
- Be capable of running the same test cases accross message topologies ranging from a single test node running in the same process as a broker, to many test nodes running on different machines, remotely connected to a broker.

Constraints:

IOP-1.	**Operating System.**	The test client scripts must run on Unix and Windows. If a test client implementation is only available on one of these platforms it only needs to run on its supported platform.
IOP-2.	**Scripting Language.**	Each test client must be startable from a Unix shell script. Tests run on Windows will use Cygwin to run these scripts. There is no need to support Windows .bat scripts.

Functional Requirements:

Introduction.

These requirements describe the behaviour of test clients for testing between different client implementations of AMQP. Each client is expected to be a single program that is capable of sending test messages to other clients and receiving and responding to test messages received from other test clients. The clients are not to be run as seperate programs for the sending and receiving parts for the sake of convenience in being able to run the clients as part of an automated build. The clients will listen for control messages broadcast by a master coordinator, to enlist them in tests, tell them which test to run, when to begin their tests, where to submit reports about the tests and when to shut down.

A centralized approach has been chosen, using a single coordinator, as test framework code which would otherwise have to be duplicated amongst all the clients will generally be put in the coordinator. The idea is to place as much logic as possible in the coordinator and as little as possible in the clients which means that code will only have to be written and maintained in one place. This code will include code for enlisting clients for tests, deciding which test case to run, and formatting and logging out the results. The alternative would be to have a de-centralized approach, where each client broadcasts the test enlist messages, finds out what other clients are available to talk to, choses which tests to run and outputs the test results. One advantage of the centralized approach, is that the coordinator should know which clients are available, and therefore which clients cannot run particular tests, or fail completely to run particular tests, and should therefore be able to log out failures for clients that fail tests in a more reliable way, than if it were up to the clients to log their own failures and ommissions.

Build tests out of a standardized construction block.

- Diagram: The test circuit.

Publisher/Receiver pair.
Each end of which is a Producer/Consumer unit.
M producers, N consumers, talking over Z destinations.

One of the stated aima of this specificiation is to "Allow tests to be posed in terms of abstract asynchronous messaging concepts that AMQP and JMS support, rather than at the level of direct interfaces". For example, we know that messages sent in a transaction, must not be delivered until the transaction is committed. This is true of AMQP as it is of JMS; as AMQP is intended to provide similar messaging semantics to JMS. The statement is also true, whether the messages are broadcast to many receivers or sent to just one.

The standard consruction block for a test, is a test circuit. This consists of a publisher, and a receiver. The publisher and receiver may reside on the same machine, or may be distributed. Will use a standard set of properties to define the desired circuit topology.

Tests are always to be controlled from the publishing side only. The receiving end of the circuit is to be exposed to the test code through an interface, that abstracts as much as possible the receiving end of the test. The interface exposes a set of 'assertions' that may be applied to the receiving end of the test circuit.

In the case where the receiving end of the circuit resides on the same JVM, the assertions will call the receivers code locally. Where the receiving end is distributed accross one or more machines, the assertions will be applied to a test report gethered from all of the receivers. Test code will be written to the assertions making as few assumptions as possible about the exact test topology.

A test circuit defines a test topology, M producers, N consumers, Z outgoing routes between them.
The publishing end of each test circuit always resides on a single JVM, even if M > 1. If publishers are to be distributed accross many machines, the test framework itself provides the scaling by running the same test circuit many times in parallel. This means that it is possible to have an arbitrary number of message publishers accross one or many machines, determined by the test setup.
The receiving half of the circuit may be local, in which case all messages come back to the same machine, or distributed in which case they may be received by many machines.
There are therefore two ways in which tests may be distributed accross multiple nodes in a network; many test circuits may be distributed and run in parallel and/or the receiving ends of those circuits may be distributed or local.
Each node in the network can play up to 2 roles in any given test; publisher or receiver. It is possible to play both roles at once, but would like to have a 'single_role' flag, that can be set to ensure that test nodes taking one role, will not participate in the other for the duration of a test. For example, in the pub /sub test want one publisher and the remaining nodes to distribute the receiver role amongst themselves.

Probing for the available test topology.

- Diagram: The available topology.

When the test distribution framework starts up, it should broadcast an 'enlist' request on a known topic. All available nodes in the network to reply in order to make it known that they are available to carry out tests. For the requested test case, C test circuits are to be run in parallel. Each test defines its desired M by N topology for each circuit. The entire network may be available to run both roles, or the test case may have specified a limit on the number of publishing nodes and set the 'single_role' flag. If the number of publishing nodes exhausts the available network and the single role flag is on, then there are no nodes available to run the receiver roles, the test will fail with an error at this point. Suppose there are P nodes available to run the publisher roles, and R nodes available to run the receiver roles. The C test circuits will be divided up as evenly as possible amongst the P nodes. The C * N receivers will be divided up as evenly as possible amongst the R nodes.

A more concrete example. There are 10 test machines available. Want to run a pub/sub test with 2 publishers, publishing to 50 topics, with 250 subscribers, measuring total throughput. The distribution framework probes to find the ten machines. The test parameters specify a concurrency level of 2 circuits, limited to 2 nodes, with the single role flag set, which leaves 8 nodes to play the receiver role. The test parameters specify each circuit as having 25 topics, unique to the circuit, and 125 receivers. The total of 250 receivers are distributed amongst the 8 available nodes, 31 each, except for two of them which get 32. The test specifies a duration of 10 minutes, sending messages 500 bytes in size using test batches of 10000 messages, as fast as possible. The distribution framework sends a start signal to each of the publishers. The publishers run for 10000 messages. The publishers request a report from each receiver on their cicruit. The receivers send back to the publishers a report on the number of messages received in the batch. The publishers assert that the correct number for the batch were indeed received, and log a time sample for the batch. This continues for 10 minutes. At the end of the 10 minutes, the publishers collate all of their timings, failures, errors into a log message. The distribution framework requests the test report from each publishing nodes, and these logs are combined together to produce a single log for the entire run. Some stats, such as total time taken, total messages through the system, total throughput are calculated and added as a summary to the log, along with a record of the requested and actual topology used to run the test.

- Diagram: The requested test applied onto the available topology.

Test Procedures.

A variety of different tests can be written against a standard test circuit, many of these will follow a common pattern. One of the aims of using a common test circuit configured by a number of test parameters, is to be able to automate the generation of all possible test cases that can be produced from the circuit combined with the common testing pattern, and an outline of a procedure for doing this is described here. The typical test sequence is described below:

A typical test sequence.

1. Initialize the test circuit from the default parameters, plus specific settings for the test.
2. Create the test circuit. The requested test parameters are applied to the available topology to produce a live circuit.
3. Send messages.
4. Request a status report.
5. Assert conditions on the publishing end of the circuit.
6. Assert conditions on the receiving end of the circuit.
7. Pass or fail the test.

The thorough test procedure.

The thorough test procedure uses the typical test sequence described above, but generates all of combinations of test parameters and corresponding assertions against the results.

The all_combinations function produces all combinations of test parameters described in Appendix A.

all_combinations : List<Properties>

The expected_results function, produces a list of assertions, given a set of test parameters. For example, mandatory && no_route -> assertions.add (producer.assertMessageReturned), assertions.add(receiver.assertMessageNotReceived).

expected_results: Properties -> List<Assertions>

For parameters : all_combinations
test_circuit = new TestCircuit(parameters).
test_circuit.start.

Send mesages.
Request status.

For assertion : exected_results(parameters)
Assert(assertion).

Common Requirements.

IOP-3.	**Directory Structure.**	All scripts to start and stop brokers and run test clients will be placed in a directory structure underneath a top-level directory called 'interop' that sits at the top level of the Qpid project.
IOP-4.	**Test Output Format.**	Output in junit xml format (because a lot of automated build software understands this format). There doesn't seem to be a schema or DTD for this format but it is simple enough. See Appendix B for an example.
IOP-5.	**Terminate On Timeout.**	Each client will keep a timeout count. Every time it gets a message it will reset this count. If it does not hear from the broker at all for 60 seconds then it will assume that the broker has died or that the other test clients are failing to communicate with it, and will terminate. Test clients will only wait on this timeout when they are actually expecting messages, for example after enlisting to a test and expecting a role assignment message, or during a test when they are expecting to be sent a test message. If neccessary, this timeout can be extended to a longer time period than 60 secods, its purpose is to ensure eventual termination of all clients during a fully automated build.
IOP-6.	**Default Virtual Host.**	All test clients will use the default virtual host (no name) for all tests, unless overriden by test parameters for a particular test case, or by command line options when starting the client.
IOP-7.	**Broadcast Control Topic.**	All test clients will listen to control messages broadcast on the routing key 'iop.control' on the default virtual host on the default topic exchange. This control topic is used for communicating with the test coordinator client.
IOP-48.	**Control Message Replies.**	All control messages broadcast by the coordinator will include a reply to field. The coordinator will listen on the reply address for responses to its control messages.
IOP-8.	**No Environment for Scripts.**	In general, start up scripts should be intelligent enough to configure the environment variables that they need in order to run. It should be sufficient to have a path configured for the neccessary run time tools (such as Java) when calling scripts. Environment variables, such as QPID_HOME, should be set by startup scripts themselves, figured out from their installation locations.

IOP-9.	**Wait Until Background Process Started.**	Scripts that start processes running in the background should not terminate until the process they are starting has succesfully started. This is neccessary for reliable testing, to ensure that subsequent scripts can be run, knowing that previous scripts have completed, with dependant proccesses in a known state. For example, it is important to start all test clients prior to starting the coordinator.

Use Case 1. Starting a Broker.

Run the broker start script.
The script starts a broker running and tries to connect to it (or otherwise ping it) until it is verified to be running.
Once the broker is verified to be running the script terminates with no error code.

Failure path: The broker fails to start or does not appear to be running after a timeout has passed. The script fails with an error code.

IOP-10.	**Broker Start Script.**	The Java and C++ brokers will define scripts that can start the broker running on the local machine, and these scripts will be located at interop/java/broker/start and interop/cpp/broker/start. The Java and C++ build processes will generate these scripts (or copy pre-defined ones to the output location) as part of their build processes.
IOP-11.	**Broker Start Failure.**	If a broker fails to start within 60 seconds its start script will timeout. Script will terminate with error code 1.
IOP-12.	**Broker Start Succesfull.**	When the broker starts succesfully the script will terminate with error code 0.

Use Case 2. Stopping a Broker.

Run the broker stop script.
The script terminates the broker that was started with the start script if it is still running.

Failure path: The broker won't terminate. The script fails with an error code.

IOP-13.	**Broker Stop Script.**	The Java and C++ brokers will define scripts that can stop the broker running on the local machine, and these scripts will be located at interop/java/broker/stop and interop/cpp/broker/stop. The Java and C++ build processes will generate these scripts (or copy pre-defined ones to the output location) as part of their build processes.
IOP-14.	**Broker Stop Timeout.**	If a broker fails to terminate within 60 seconds its stop script will timeout. Script will terminate with error code 1.
IOP-15.	**Broker Stop Succesfull.**	When the broker stops succesfully the script will terminate with error code 0.

Use Case 3. Starting a Test Client.

Run the client start script. The caller will pass in the address of the broker to connect to.
The script starts a client running.
The client starts running but waits for further instruction before running its tests.
The start script will terminate but leave the client running as a forked process.

Failure path: The client will not start, or fails to connect to the specified broker. The script will terminate with error code 1.

IOP-16.	**Client Start Scripts.**	For each client implementation, <client>, there will be a start script located at interop/<client>/client/start. The build processes for each client will generate these scripts and output them to this location as part of their build process.
IOP-17.	**Client Start Timeout.**	If the client fails to start and connect to the specified broker within 60 seconds the script will terminate with error code 1.
IOP-18.	**Client Start Succesfull.**	When the client starts successfully its script will terminate with error code 0.
IOP-19.	**Client Start Broker and Port.**	The -b <hostname> option will be used to instruct the start script to connect to the specified hostname. The -p <port> option will similarly allow the port to be specified.
IOP-20.	**Client Virtual Host.**	The default virtual host to connect to, may be overridden with the -v <virtual_host> command line option, which will be accepted by all test clients.
IOP-21.	**Client Start General Parameters.**	General parameters may be passed to the client start scripts using the synax name=value. These name/value pairs may be used by specific test cases to override default test parameters. See Appendix C for a list of test parameters.

Use Case 4. Starting the Coordinator.

- The requirements defined for Use Case 3, also apply to this use case.

Run the testall start script. The caller will pass in the address of the broker to connect to.
The script starts the coordinator client running.
The coordinator will manage the test procedure.
The scipt will terminate when the coordinator has completed.

Failure path: The coordinator will not start, or fails to connect to the broker. The script will terminate with error code 1.

IOP-22.	**Coordinator Test Script.**	There will be a coordinator test script that kicks off the testing process once all clients have been started. It is to be located at interop/testall. It will start a coordinator test client that issues test invites, assigns roles, collects results and terminates test clients when all tests have been run.

Use Case 5. Overall Test Procedure.

Start a broker running using its start script as described by Use Case 1.
Call the start all clients script on each of the machines where there are clients that are to be tested. The caller will pass in address of the broker to connect to, and any additional parameters.
The start all script will scan for all start scripts located under interop/<client>/client/start and call each of them forwarding its command line arguments on the call. This performs Use Case 3 for each client.
Call the coordinator test client script. This is described as Use Case 4.

The coordinator test script will broadcast an invite message, with no test name on the control topic. The lack of a test name indicates that this is a compulsory invite, to which all clients must enlist.
Each client will respond with an enlist message. This message will contain the routing key on the default topic exchange to which the client has bound its private control queue.
The coordinator retains the list of available clients, and the addresses of their control queues.

The coordinator will broadcast an invite to a named test. This invite may also contain any parameters needed to configure the test, that are relevant to a clients choice to accept the invite or not.
All clients that are able to participate in this test will reply to the invite with enlist messages. Clients may opt to participate in the test depending on the test parameters, if desired.
The coordinator will send messages to assign roles to the sender and receivers private control topics. These messages will contain the test parameters and roles. The test parameters may also include additional parameters not in the original invite, for test parameters that are to be set on a per test instance basis.
The clients will respond with accept role messages.
The coordinator will wait until it has received acceptances from both roles.
The coordinator will issue a start message to the client with the sender role.
The sender client will send its test messages. Once the test has completed the sender will send a report message to the coordinator, giving details about the message that it sent.
The coordinator will wait until it receives a report message from the sender.
The coordinator will issue a status request message to the reciever role.
The receiver will reply with a report, giving details about the messages it has received.
The coordinator will wait until it receives a report message from the receiver.
The coordinator will compare the sender and receiver reports in order to decide whether the test passed or failed.
The coordinator will check its list of available clients and log out failures for any combinations of clients that were not tested because they did not enlist for the test.

Once all test cases are complete, the coordinator will broadcast a shutdown message.
All clients will terminate on receipt of the shutdown message.
The coordinator will terminate.
Terminate the broker using its stop script.

IOP-23.	**Start All Script.**	There will be a start all clients script, located at interop/startall. The startall script finds all client starts scripts under interop/<client>/client /start and calls them.
IOP-24.	**Start All Script Options Forwarding.**	The start all script will take the same command line options as the client start scripts and will pass these command line options on to them.

IOP-25.	**Invite Message.**	For every test case the coordinator will broadcast an invite message on the control topic. This message will be identified by the header field, "CONTROL_TYPE", having the value, "INVITE". This message will also include the name of the test case and may also include some test parameters. (See IOP-48, for the reply to address.)
		`"CONTROL_TYPE", "INVITE"`
		`"TEST_NAME", "<test_case>"`
		`... optional test parameters.`

IOP-26.	**Initial Invite.**	At the start of the test procedure the coordinator will broadcast a compulsory invite, to which all available clients must enlist, in order to declare their availability and to enable the coordinator to detect when there are clients that did not participate in some tests. The compulsory invite will be differentiated form an ordinatory invite because it will have no "TEST_NAME" header field.

IOP-27.	**Enlist Message.**	Every test client that receives an invite message will respond by declaring its availability to run interop tests. The client will send an enlist message by replying to the invite message. The enlist message will be identified by the header field, "CONTROL_TYPE", having the value, "ENLIST". The client will declare the routing key on which it expects to be sent private control messages. The client will also declare a unique name by which it can be identified (see IOP-35). The declare available message will contain the following header fields with this information:
		`"CONTROL_TYPE", "ENLIST"`
		`"CLIENT_NAME", "<client_name>" (see IOP-35 for rules about the client name).`
		`"CLIENT_PRIVATE_CONTROL_KEY", "iop.control.<client_name>" (see IOP-36)`

IOP-28.	**Assign Role Message.**	Having selected clients to participate in a particular test case, the coordinator will send those clients messages to assign the roles they will play in the test case, on the clients private control topics. Each test case has sender and receiver roles. This message will be identified by the header field, "CONTROL_TYPE", having the value, "ASSIGN_ROLE". The full test parameters will be included in this message, allowing tests to be configured on a per test instance basis.
		`"CONTROL_TYPE", "ASSIGN_ROLE"`
		`... full test parameters.`

IOP-29.	**Accept Role Message.**	A client receiving an assign role message, will reply to it with an accept role message. This message also indicates that the client is ready to start the test. This message will be identifier by the header field, "CONTROL_TYPE", having the value, "ACCEPT_ROLE". `"CONTROL_TYPE", "ACCEPT_ROLE"`
IOP-30.	**Start Message.**	The coordinator will send a start message to begin the test procedure. All test clients will listen for this message on their private control topics. The start message will be identified by the header field, "CONTROL_TYPE", having the value, "START". `"CONTROL_TYPE", "START"`

IOP-31.	**Report Message.**	Once the test clients have completed a test case, they will send the coordinator a report about the actions they have performed. In the case of senders, this report will be sent once they have finished sending test messages. In the case of receiver, this report will be sent in response to a status request from the coordinator (see IOP-49). The report message will be identified by the header field, "CONTROL_TYPE", having the value, "REPORT". Its message body, or additional header fields will contain the report, specific to the test case being run.
		`"CONTROL_TYPE", "REPORT"`
		`... test specific parameters.`
		`Message body, Test case specific report.`

IOP-49.	**Status Request Message.**	Once the coordinator has received the senders report, it will send a status request to the receiver, to request the receivers report. This message will be identified by the header field, "CONTROL_TYPE", having the value, "STATUS_REQUEST". `"CONTROL_TYPE", "STATUS_REQUEST"`
IOP-34.	**Terminate Message.**	The coordinator will wait for all test clients to complete their tests for all test cases at which time it will broadcast a terminate message to the control topic. The terminate message will be identified by the header field, "CONTROL_TOPIC", having the value, "TERMINATE". Upon receipt of this message the test clients will terminate. `"CONTROL_TYPE", "TERMINATE"`
IOP-35.	**Client Name.**	Each test client will provide a unique name for itself that reflects its implementation language and distinguishes it from the other clients. Clients may append an environment identifier onto this name to cater for the case where the same client is used multiple times in an interop test. For example, the same client might be run on two different operating systems, in order to check that it works correctly on both. Example names in this case might be "java-win" and "java-linux".
IOP-36.	**Private Client Control Topic.**	Each test client will listen for test control messages directed specifically to it on the default topic exchange. The routing key for these messages will consist of "iop.control." followed by the client name (see IOP-35). A topic exchange is used, rather than a direct exchange, to cater for the situation where multiple instances of a client are run in parallel and tests are to be scaled accross many clients (not currently in scope, see Waiting Room). It also allows a listener to be attached to the default topic exchange to listen to all control messages using a wildcard selector.

IOP-37.	**Seperate Connection for Control Topic.**	Test clients should create open a seperate connection to communicate with the control topics on the default topic exchange, to that which they use to perform tests. This is so that a channel level error that results in the closing of a connection during a test, may still allow a client to succesfully send a failure report to the coordinator.

Common Requirements for Test Cases.

Test cases that use these requirements mention them in the description of the test case.

IOP-38.	**Message Counts.**	Whenever a test client recieves a message from another test client it will increment the total count of messages received from that client. Test messages will contain the name of the sending client in the header field "CLIENT_NAME", and the count will be held against a combination of that name and the messages correlation id (see IOP-42).
IOP-39.	**Message Count Reset.**	Whenever a test client is begining a new test case (when it accepts a role) it will reset its message counts to zero.

IOP-41.	**Message Count Report Message.**	Upon receipt of a status request message, a test client will reply with a report message. The report message will be identified by the header field "CONTROL_TYPE", having the value, "REPORT" (as described by IOP-31). In addition to this, the header field, "MESSAGE_COUNT" will contain the count of messages received since the last reset as a signed 32-bit integer.
		`"CONTROL_TYPE", "REPORT"`
		`"MESSAGE_COUNT", <count> (signed 32 bit integer)`

IOP-42.	**Correlation Id.**	When sending test messages, clients will identify all messages using a unique correlation id for the test case instance. This will differentiate test messages in a situation where the same client is scaled up to run a test case many times in parallel (not in scope, see Waiting Room).
IOP-43.	**Test Connections.**	Test clients will create connections to send test messages on when they are assigned roles. In many cases this will consist of creating a single connection, and a producer or consumer for the test routing key or queue. In some tests, which simulate the activity of many message receivers, multiple connections may be opened.

Test Case 1. Dummy Run.

The sending client will not send any test messages at all. It will send a report message on the control topic, declaring that the test has passed.

The purpose of this test case is to check that clients can interoperate succesfully with the test coordinator and participate in the sequencing of the tests.

IOP-50.	**Test Case 1 Name.**	The "TEST_NAME" field in the test invite (IOP-25) will be "TC1_DummyRun" for this test.

Test Case 2. Basic P2P Test.

- This test case uses requirements IOP-38 to 43 inclusive.

The sending client creates a fresh correlation id, and the entire test case conversation uses this id.
The sending client will send the required number of test messages to the test routing key on the default direct exchange.
The sending client will send a message count report to the coordinator.
In response to a status request from the coordinator, the receiving client will reply with a message count report.
The coordinator will compare the messages received to the messages sent and pass or fail the test accordingly.

IOP-44.	**Basic P2P Setup.**	Prior to assigning roles, the coordinator will bind a queue to the default direct exchange with a routing key, the same as the queue name. It will create a fresh queue and key for every test case instance.

IOP-45.	**Basic P2P Assign Role Parameters.**	In addition to the invite message format defined in IOP-26, the basic p2p test invite will also include the following parameters.
		`"P2P_QUEUE_AND_KEY_NAME", "<name>"`
		`"P2P_NUM_MESSAGES", <count> (signed 32 bit int),`
		`P2P_NUM_MESSAGES property.`

IOP-51.	**Test Case 2 Name.**	The "TEST_NAME" field in the test invite (IOP-25) will be "TC2_BasicP2P" for this test.

Test Case 3. Basic Pub/Sub Test.

- This test case uses requirements IOP-38 to 43 inclusive.

The sending client creates a fresh correlation id, and the entire test case conversation uses this id.
The sending client will send the required number of test messages to the test routing key on the default topic exchange.
The sending client will send a message count report to the coordinator.
In response to a status request from the coordinator, the receiving client will reply with a message count report. This number will be the number of messages sent multiplied by the number of receivers being simulated by the receiving client.
The coordinator will compare the messages received to the messages sent and pass or fail the test accordingly.

IOP-46.	**Basic Pub/Sub Setup.**	Prior to assigning roles, the coordinator will choose a routing key for the test. If will create a fresh key for every test case instance.

IOP-47.	**Basic Pub/Sub Invite Parameters.**	In addition to the invite message format defined in IOP-26, the basic pub/sub test invite will also include the following parameters.
		`"PUBSUB_KEY", "<key>"`
		`"PUBSUB_NUM_RECEIVERS", <count> (signed 32 bit int), PUBSUB_NUM_RECEIVERS property.`
		`"PUBSUB_NUM_MESSAGES", <count> (signed 32 bit int), PUBSUB_NUM_MESSAGES property.`

IOP-52.	**Test Case 3 Name.**	The "TEST_NAME" field in the test invite (IOP-25) will be "TC3_BasicPubSub" for this test.

Test Case 4. P2P Test with Different Message Sizes.

- This test case uses requirements IOP-38 to 43 inclusive.

The sending client creates a fresh correlation id, and the entire test case conversation uses this id.
The sending client will send the required number of test messages to the test routing key on the default direct exchange.
The sending client will send a message count report to the coordinator.
In response to a status request from the coordinator, the receiving client will reply with a message count report.
The coordinator will compare the messages received to the messages sent and pass or fail the test accordingly.
The above test cycle will be repeated for each message size to test.

IOP-53.	**P2P Message Size Test Setup.**	Prior to assigning roles, the coordinator will bind a queue to the default direct exchange with a routing key, the same as the queue name. It will create a fresh queue and key for every test case instance.

IOP-54.	**P2P Message Size Test Assign Role Parameters.**	In addition to the invite message format defined in IOP-26, the basic p2p test invite will also include the following parameters.
		`"P2P_QUEUE_AND_KEY_NAME", "<name>"`
		`"P2P_NUM_MESSAGES", <count> (signed 32 bit int), P2P_NUM_MESSAGES property.`
		`"messageSize", <count> (signed 32-bit int).`

IOP-55.	**P2P Message Size Test Sizes**	The following values for the message size parameter will be tested: 0K, 63K, 64K, 65K, 127K, 128K, 129K, 255K, 256K, 257K.
IOP-56.	**Test Case 4 Name.**	The "TEST_NAME" field in the test invite (IOP-25) will be "TC4_P2PMessageSize" for this test.

Test Case 5. Pub/Sub Test with Different Message Sizes.

- This test case uses requirements IOP-38 to 43 inclusive.

The sending client creates a fresh correlation id, and the entire test case conversation uses this id.
The sending client will send the required number of test messages to the test routing key on the default topic exchange.
The sending client will send a message count report to the coordinator.
In response to a status request from the coordinator, the receiving client will reply with a message count report. This number will be the number of messages sent multiplied by the number of receivers being simulated by the receiving client.
The coordinator will compare the messages received to the messages sent and pass or fail the test accordingly.
The above test cycle will be repeated for each message size to test.

IOP-57.	**Pub/Sub Message Size Test Setup.**	Prior to assigning roles, the coordinator will choose a routing key for the test. If will create a fresh key for every test case instance.

IOP-58.	**Pub/Sub Message Size Test Invite Parameters.**	In addition to the invite message format defined in IOP-26, the basic pub/sub test invite will also include the following parameters.
		
		`"PUBSUB_KEY", "<key>"`
		`"PUBSUB_NUM_RECEIVERS", <count> (signed 32 bit int), PUBSUB_NUM_RECEIVERS property.`
		`"PUBSUB_NUM_MESSAGES", <count> (signed 32 bit int), PUBSUB_NUM_MESSAGES property.`
		`"messageSize", <count> (signed 32-bit int).`

IOP-59.	**P2P Message Size Test Sizes**	The following values for the message size parameter will be tested: 0K, 63K, 64K, 65K, 127K, 128K, 129K, 255K, 256K, 257K.
IOP-60.	**Test Case 5 Name.**	The "TEST_NAME" field in the test invite (IOP-25) will be "TC5_PubSubMessageSize" for this test.

Waiting Room:

Contains ideas for possible future directions relating to this spec.

Command processor. Test cases to be written using a command language (perhaps in XML) on top of a common client API. Interpreter for this to be implemented using each client library. Test cases need only be written once and can be run by the interpreters. Command language rich enough to exercise the whole AMQP protocol. May not handle client specific edge cases. Good for ensuring test consistency, but may take a fair amount of time to do.

How I anticipate this being run as part of a fully automated build. Will try to get a free licence for Anthill Pro 3 as they offer free licences for open source projects. Viewtier Parabuild is another possibility. Anthill Pro runs a central build server that does all its work through build agents that can run on many boxes. It also lets you define build workflows. I imagine running a Unix agent to build the c++, java and python stuff, and a Windows agent for the .net stuff. Will define a workflow that starts a broker on the unix box, then starts all clients built on the unix and windows boxes in parallel, then runs the entire test procedure across all clients, then terminates the broker on the unix box. The agents send back the test results to the central server.

Full testing of field tables. Make sure that every possible data type is tested and confirmed to encode and decode correctly between all client implementations.

Testing more of the protocol. Add tests to more fully exercise the complete AMQP protocol.

Allow scaling of test clients. Each test client should only be run once (in each environment) and they create unique names for themselves. Tests are only run between pairs of single clients, with a single sender and number of receivers defined by the test case (often 1). Clients listen for control messages on topics, and use correlation id's in all tests messages to differentiate themselves, were multiple senders to be active. This has been done deliberately to allow for future expansion of the test framework to allow scaling up of the tests by starting more clients in parallel on the same environment. To do this each client might also create a sequence number, to unqiuely identify itself, as the client names will no longer be unique. Reports from senders will include client name, sequence number and correlation id. Status requests to receivers may specify client name, sequence number and correlation id to get specific reports, or ommit correlation id or sequence number to get a bulk report of all messages with a particular client.

More sophisticated reporting. Message count reports are fairly minimal. Might also put an entire list of messages send/recieved in a report, in order to check that there were no ommissions or duplicates.

Appendix A, General Notes:

Brokers that need to be interop tested: C++ and Java

Clients that need to be interop tested: C++ , Java, Java 1.4 retrotranslation, C++, .Net 2.0, .Net 1.1, (Mono?), Python, Ruby.

Appendix B, Example of XML Format for Test Ouput:

I don't think there is a DTD or schema for this but the XML output from JUnit looks like the example below. This is a convenient choice for the output format from these test results even if the code does not actually use JUnit (or cppunit or nunit) iternally, because automated build servers generally understand and are able to produce test reports from it.

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<testsuite errors="0" skipped="0" tests="18" time="0.02" failures="0" name="org.apache.qpid.framing.
BasicContentHeaderPropertiesTest">
  <properties>
    <property value="Java(TM) 2 Runtime Environment, Standard Edition" name="java.runtime.name"/>
    ... (there were lots of properties).
  </properties>
  <testcase time="0.02" name="testRejectedExecution"/>
  ... (there were lots of test cases).
</testsuite>
```

Appendix C, Test Parameters.

	Possible Values	Default Value
Connection properties.		
broker	tcp, vm	tcp://localhost
vhost		<empty>
username		guest
password		guest
Topology properties.		
max_publishing_node		1
single_role	true, false	true
Circuit properties.	Total: 2^2 = 4 combinations.	
num_publishers		1
num_consumers		1
num_destinations		1
base_out_route_name		ping
base_in_route_name		pong
bind_out_route	true, false	true
bind_in_route	true, false	false
consumer_out_active	true, false	true
consumer_in_active	true, false	false
JMS flags and options.	Total: 2 * 2 * 2 * 6 = 48 combinations.	
transactional	true, false	false
persistent	true, false	false
no_local	true, false	false
ack_mode	tx, auto, client, dups_ok, no_ack, pre_ack	auto
AMQP/Qpid flags and options.	Total: 2^4 = 16 combinations.	
exclusive	true, false	false
immediate	true, false	false
mandatory	true, false	false
durable	true, false	false
prefetch_size		
header_fields		
Standard test parameters.	Total: 3 combinations.	

message_size	no_body, one_body, multi_body	one_body
num_messages		100
outgoing_rate		
inbound_rate		
timeout		30 seconds
tx_batch_size		100
max_pending_data		

Total combinations over all test parameters: 4 * 48 * 16 * 3 = 9216 combinations.

Defaults give an in-VM broker, 1:1 P2P topology, no tx, auto ack, no flags, publisher -> receiver route configured, no return route.

Appendix D, Command line options.

IOP-21 states that general parameters can be passed on the command line using name=value syntax. The coordinator understands the following parameters, and will use them to override the default values for the tests. Individual test cases refer to the command line parameter that they take their test parameters from.

Parameter	Default
P2P_NUM_MESSAGES	50
PUBSUB_NUM_RECEIVERS	5
PUBSUB_NUM_MESSAGES	10

Appendix E, Clock Synchronization Algorithm.

On connection/initialization of the framework, synch clocks between all nodes in the available toplogy. For in vm tests, the clock delta and error will automatically be zero. For throughput measurements, the overall test times will be long enough that the error does not need to be particularly small. For latency measurements, want to get accurate clock synchronization. This should not be too hard to achieve over a quiet local network.

After determining the list of clients available to conduct tests against, the Coordinator synchronizes the clocks of each in turn. The synchronization is done against one client at a time, at a fairly low messaging rate over the Qpid broker. If needed, a more accurate mechanism, using something like NTP over UDP could be used. Ensure the clock synchronization is captured by an interface, to allow better solutions to be added at a later date. Here is a simple algorithm to get started with:

1. Coordinator tells client to synchronize its clock with the coordinators time.
2. Client stamps current local time on a "time request" message and sends to Coordinator.
3. Upon receipt by Coordinator, Coordinator stamps Coordinator-time and returns.
4. Upon receipt by Client, Client subtracts current time from sent time and divides by two to compute latency. It subtracts current time from Coordinator time to determine Client-Coordinator time delta and adds in the half-latency to get the correct clock delta.
5. The first result should immediately be used to update the clock since it will get the local clock into at least the right ballpark.
6. The Client repeats steps 1 through 3, 25 or more times, pausing a few tens of milliseconds each time.
7. The results of the packet receipts are accumulated and sorted in lowest-latency to highest-latency order. The median latency is determined by picking the mid-point sample from this ordered list.
8. All samples above approximately 1 standard-deviation from the median are discarded and the remaining samples are averaged using an arithmetic mean.

The above algorithm includes broker latency, two network hops each way, plus possible effects of buffering/resends on the TCP protocol. A fairly easy improvement on it might be:

1. Coordinator tells client to synchronize its clock with the coordinators time, provides a port/address to synchronize against.
2. Clients sends UDP packets to the Coordinators address and performs the same procedure as outlined above.

Appendix F, Deleted Requirements:

Put deleted requirements here, in case they can be re-used.

IOP.	**Client Start Messages Per Test.**	The -m <num_messages> option will be used to tell the client how many messages to send per test.
IOP.	**Client Number of Receivers.**	For topic testing each client will simulate the behaviour of many clients listening to the same topic. The number of receivers per test client for topic tests will be sepcified by the -r <num_receivers> command line option.

IOP.	**Client Default P2P Test Direct Key.**	Each test client will listen for test messages on the default direct exchange. The routing key for these messages will consist of the client name (see IOP-35) followed by ".direct".
IOP.	**Client Default Pub/Sub Test Direct Key.**	Each test client will listen for test messages on the default topic exchange. The routing key for these messages will consist of the client name (see IOP-35) followed by ".topic".
IOP.	**Test Done Message.**	Once a test client has completed its role, it will send the coordinator a test done message on the control topic. This message will be identified by the header field, "CONTROL_TYPE", having the value, "TEST_DONE". The client will also post its name in the "CLIENT_NAME" header field. `"CONTROL_TYPE", "TEST_DONE"`
IOP.	**End Role Message.**	Once the coordinator receives a report for a test case, it will send end role messages to the private control topics of all clients participating in the test case. This message will be identified by the header field, "CONTROL_TYPE", having the value, "END_ROLE". `"CONTROL_TYPE", "END_ROLE"`

IOP.	**Client Status Request Message.**	When a test client has completed sending test messages it may request the count of actual messages receieved from the test client to which it sent the messages. The status request message will be send to the receving test clients individual control topic. This message will be identified by the header field, "CONTROL_TYPE", having the value, "STATUS_REQUEST", and will contain the name of the sending client in the header field "CLIENT_NAME".
		`"CONTROL_TYPE", "STATUS_REQUEST"`
		`"CLIENT_NAME", "<client_name>"`