## Message Node

### Description of Component

The Message Node provides Real Time Message Oriented communication functionalities used by Hyperties to communicate with each other. It provides different communication patterns including publish/subscribe communication and Request/Response communication.

### Methodology and setup

For the evaluation of the individual Message Nodes (MNs) a set of Karma[10] tests has been developed. These common tests are executed against each Message Node implementation and the results are documented.

The set of tests has been updated to the latest specification changes and at some points slightly extended due to new features compared with the tests that have been described in the previous Deliverable D6.3.

The first set of tests evaluates the conformance of the MNs with the specified request/response patterns of the message API. The second set of tests evaluates the performance of the MNs in forced situations of increased load.

The individual tests are described in the following chapter.

In order to be comparable all tests have been performed on a single machine with locally deployed Message Nodes and a locally deployed Domain Registry. (This is the same setup that has been used for the tests in the previous Assessment report D6.3)

The used machine has following parameters:

* CPU: Intel Core i7-4712MQ, 2.30GHz (8 Cores)
* 8 GB RAM
* SSD
* OS: Debian (stretch)

The Domain Registry was running in its own Docker container in a custom Docker network, exposing the internal port 4567 to the host port 4567. So it was accessible from other docker containers in the same custom network as well as from the host via this port.

The individual MNs have been deployed from their pre-built docker-images available on the rethink docker-hub. All MNs have been operated as single nodes in the simpliest possible configuration. Potentially available cluster modes were not used.

### Conformance metrics

These tests evaluate the conformance of the MNs with the specified message patterns. Example requests are created according to the specification at [13] and sent to the Message Node. The corresponding responses are then compared with the expected results. Performance parameters, like the response time, are not evaluated in these tests. The following sub-sections give a short description of the individual conformance tests.

#### Stub connection and disconnection

The purpose of this test is to ensure that the Stub exposes the "status" events as specified in the Protocolstub state machine at <https://github.com/reTHINK-project/specs/blob/master/messaging-framework/protostub-state-machine.png>.

It instantiates a stub and performs a connection and then a disconnection of the stub to the corresponding MN. The tests check that the stubs are generating the correct status event sequence (“created”, “in-progress”, “live”, “disconnected”) for each connection/disconnection situation.

|  |  |
| --- | --- |
| Test 1.1 | Stub connect/disconnect |

#### Hyperty address allocation messages

The purpose of this test is to ensure the Conformance of the MN operations with the message specification for hyperty address allocationshttps://github.com/reTHINK-project/dev-service-framework/blob/master/docs/specs/messages/address-allocation-messages.md.

It checks the allocation and de-allocation of hyperty addresses by the MNs. It includes following sub-tests:

|  |  |
| --- | --- |
| Test 2.1 | allocation of a single address |
| Test 2.2 | de-allocation of a single address |
| Test 2.3 | allocation of an address block of 3 addresses |
| Test 2.4 | de-allocation of an address block of 3 addresses by given address array |
| Test 2.5 | allocation of an address block with a given “allocationKey” as identifier for this block of addresses |
| Test 2.6 | de-allocation of an address block identified by a given allocationKey |

#### Object address allocation messages

The purpose of this test is to ensure the Conformance of the MN operations with the message specification for object address allocations.

It checks the allocation and de-allocation of object addresses by the MNs. It includes following sub-tests:

|  |  |
| --- | --- |
| Test 3.1 | allocation of a single address incl. check that a given "scheme" is used for the object address allocations |
| Test 3.2 | de-allocation of a single address |
| Test 3.3 | allocation of an address block of 3 addresses |
| Test 3.4 | de-allocation of an address block of 3 addresses by given address array |
| Test 3.5 | allocation of an address block with a given “allocationKey” as identifier for this block of addresses |
| Test 2.6 | de-allocation of an address block identified by a given “allocationKey” |

#### Subscription and object update messages

The purpose of this test is to ensure the Conformance of the MN operations with the Data synchronization message specification.

It checks the subscription and un-subscription for given object addresses at the MN as well as the correct publication of object update events to the subscribers. It includes following sub-tests:

|  |  |
| --- | --- |
| Test 4.0 | Test setup: allocation of an object address |
| Test 4.1 | subscription for an object address with a given body.source attribute (the given body.source must be used as subscriber url) |
| Test 4.2 | subscription for an object address without a body.source attribute (the from address of the subscribe message must be used as subscriber url) |
| Test 4.4 | update of the subscribed object by the reporter and expect correct update events on both subscribers |
| Test 4.4 | unsubscription of both subscribers, expecting correct 200 OK response |
| Test 4.5 | another update of the subscribed object and expect that no events are received by the (now unsubscribed) subscribers |

#### Registration messages

The purpose of this test is to ensure the Conformance of the MN operations with the Registration message Specification [11].

This set of tests requires a running Domain Registry component that is accessible by the MNs. The MN uses a RegistryConnector component to forward the received messages to the Domain Registry and receive back the responses.

It includes following sub-tests:

|  |  |
| --- | --- |
| Test 5.1 | Test setup: registration of an allocated hyperty address for a given userid |
| Test 5.2 | Search a hyperty by given userid |
| Test 5.3 | Search a hyperty address by given userid and scheme |
| Test 5.4 | Search a hyperty by given userid and resource-types |
| Test 5.5 | Search a hyperty by given userid, scheme and resource-types |
| Test 5.6 | Search and retrieve hyperty-data by a given hyperty address |
| Test 5.7 | Search for a non existing registry object/hyperty → expect correct error message |
| Test 5.8 | Test correct handling of keep-alive messages for an active registration |
| Test 5.9 | Subscription for update events, when the status of the registered object changes |
| Test 5.10 | Test delivery and receiving of status update events (very NEW in spec.) |
| Test 5.11 | unregistration of a hyperty address |

### Performance metrics

These tests evaluate the performance and robustness of the MNs in forced situations of increased load. Therefore requests are sent in loops of increasing iteration counts and with differing message sizes as fast as possible. For each of these tests, the duration is measured and used as metric for the evaluation.

Each test is executed 3 times. The results in the table are the mean values of the 3 test runs. Messages are sent out in iterations as fast as possible without any delays between messages to put the MN under stress.

#### Hyperty address allocation messages

These tests create and send an increasing number of allocation and de-allocation messages for Hyperty addresses to the MNs and expect at each time a correct 200 OK response. The sizes of these messages are already defined by the specification. They can not be smaller than specified and it also makes no sense to artificially increase their size. Therefore only the number of iterations and the number of addresses to allocate vary. Furthermore the tests with several address numbers are executed with and without an “allocationKey” parameter, which can be used to identify blocks of allocated addresses later on (e.g. for an de-allocation).

Also the time for the de-allocation of all addresses is measured.

Following measurements are performed in this set of test:

* hyperty address allocation and de-allocation requests for 1 address each for 100, 1000 and 10000 iterations
* hyperty address allocation and de-allocation requests for a block of 3 address each without an “allocationKey” for 100, 1000 and 10000 iterations
* hyperty address allocation and de-allocation requests for a block of 3 address each with “allocationKey” for 100, 1000 and 10000 iterations

#### Object address allocation messages

These tests repeat the same procedure as previously described for hyperty addresses now for object addresses too. The same comments and notes apply here as well.

Following measurements are performed in this set of tests:

* Object address allocation and de-allocation requests for 1 address each for 100, 1000 and 10000 iterations
* Object address allocation and de-allocation requests for a block of 3 address each without “allocationKey” for 100, 1000 and 10000 iterations
* Object address allocation and de-allocation requests for a block of 3 address each with “allocationKey” for 100, 1000 and 10000 iterations

#### Hyperty messages

These tests send an increasing number of messages from one allocated Hyperty to a second allocated Hyperty. Each Hyperty is connected via its own stub. This simulates the behaviour of messaging between Hyperties which are deployed in two different Runtimes.

During the tests the payload sizes are increased from 100B over 1kB to 10kB. The purpose of this test is to ensure that all messages arrive in the same order they were sent and to measure the overall needed time.

The measured time frame starts when the first message from Hyperty 1 via Stub 1 is sent and stops when the last message is received by Hyperty 2 via Stub 2.

Following measurements are performed:

* Message with payload of 100B for 100, 1000 and 10000 iterations
* Message with payload of 1kB for 100, 1000 and 10000 iterations
* Message with payload of 10kB for 100, 1000 and 10000 iterations

#### Subscription and Publication of Object updates

These tests send a publish Object update message from one reporter to a increasing number of subscribers. During the tests also the payload sizes are increased from 100B over 1kB to 10kB.

For this purpose a single reporter Hyperty is connected to the MN via a first stub and an increasing number of subscriber hyperties are connected via their own stubs, one stub for each subscriber hyperty. In the preparation phase, the reporter stub allocates an object address first and then all subscriber Hyperties send subscription messages for updates on this object and wait for their OK response.

The measured time frame starts when the Reporter Hyperty sends the update message and stops when all subscribers have received the corresponding update message. The order in which the subscribers receive the update message is not considered as test criteria.

Following measurements are performed:

* Update Message with payload of 100B for 100, 200, and 1000 subscribers
* Update Message with payload of 1kB for 100, 200, and 1000 subscribers
* Update Message with payload of 10kB for 100, 200, and 1000 subscribers

### Summary of MN Assessment

#### Conformance test results

The following figure summarizes the results of the conformance tests for all 3 Message Nodes.

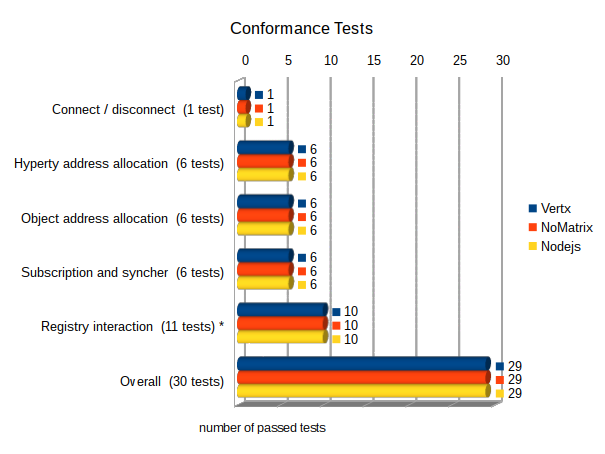


Figure 1: Overview of specification conformance test results

***Connection and status event emission test***

All 3 nodes pass the basic connect and disconnect test and emitted the correct sequence of status events during the deployment and connection phase.

***Address allocation / de-allocation tests:***

Also all tests for the allocation and de-allocation of hyperty and object addresses where passed successfully by all 3 message nodes.

***Subscription and publish:***

All 3 nodes have passed the test for subscription and publication of data object updates according to the latest specification

***Registry interactions:***

**Notes:**

For the interaction with the Domain Registry all 3 nodes make use of a RegistryConnector component that is available as a plug-in module. Therefore the pure message format and exchange between the MN and the domain registry is identical for all 3 MNs. Differences are only be caused by the way how messages are identified and assigned to the RegistryConnector plugin and how responses are parsed and routed back.

**Results:**

All MNs are failing to pass Test 5.10 (Test delivery and receiving of status update events). This test checks a quite recent addition of the spec. which allows any runtime to subscribe for status-changes of hyperties or objects in the domain registry. This functionality will require an architectural extension of the RegistryConnector, because it needs to implement a server interface as well in order to receive notification from the domain registry. So far it only acts in the client role, sending requests and to the domain registry and expecting responses. This extension is – at the time of test execution - not yet implemented in the RegistryConnector and therefore the corresponding test fails for all nodes using it.

The mentioned new feature is not yet part of the rethink core framework – therefore this failed test is not crucial for the use of the MNs.

All 3 Nodes are passing the remaining 10 tests successfully.

#### Performance test results

***Allocation and de-allocation of Hyperty addresses***

The following 2 figures provide a summary of the performance of the MNs in terms of allocation and de-allocation of hyperty addresses.

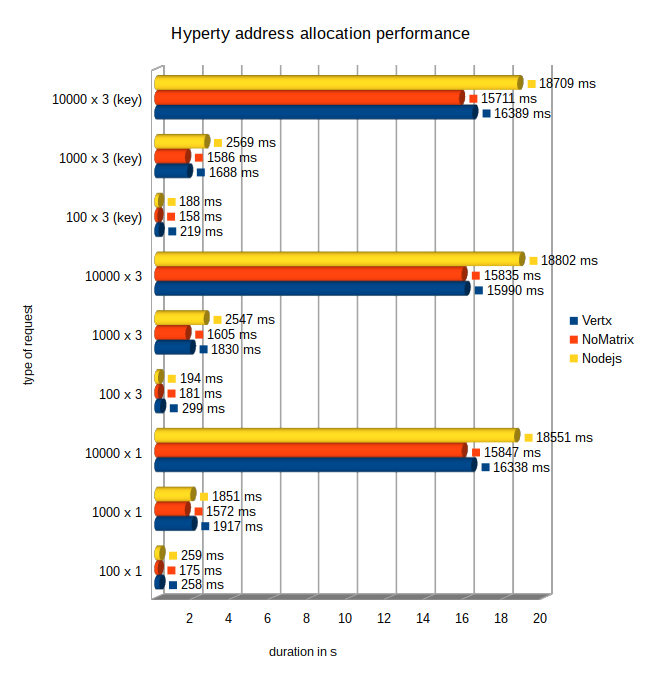


Figure 2: Performance of Hyperty address allocations

All 3 nodes successfully passed all hyperty address allocation tests. The performance of the hyperty allocations is on a comparable level for all nodes, while the NoMatrix MN performs slightly faster than the Vertx and Nodejs MNs. The NodeJS MN became a little bit slower especially for higher iteration numbers and for tests with several address allocations at once.

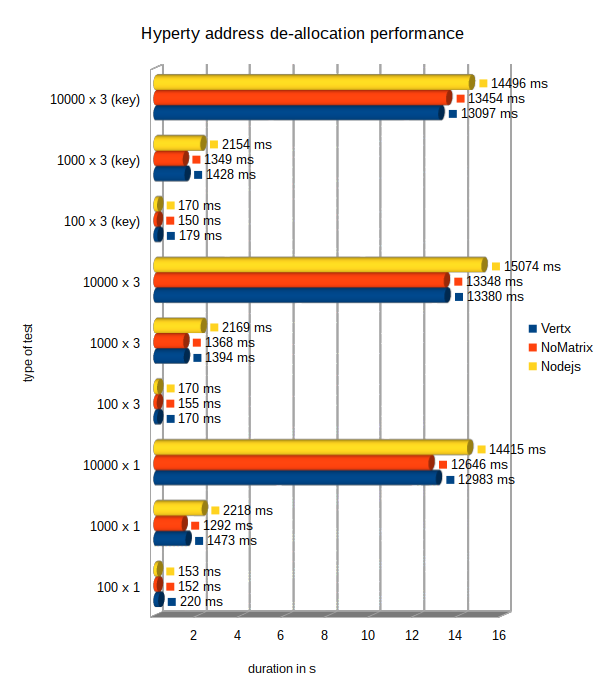


Figure 3: Performance of Hyperty address de-allocations

All 3 nodes passed successfully all hyperty address de-allocation tests. The performance of the de-allocations is in general up to 20% better than for the hyperty address allocations.

The NoMatrix MN performs slightly faster than the Vertx and Nodejs MNs. The NodeJS MN became a little bit slower especially for higher iteration numbers and for tests with several address de-allocations at once.

***Allocation and de-allocation of Object addresses***

The following 2 figures provide a summary of the performance of the MNs in terms of allocation and de-allocation of Object addresses.

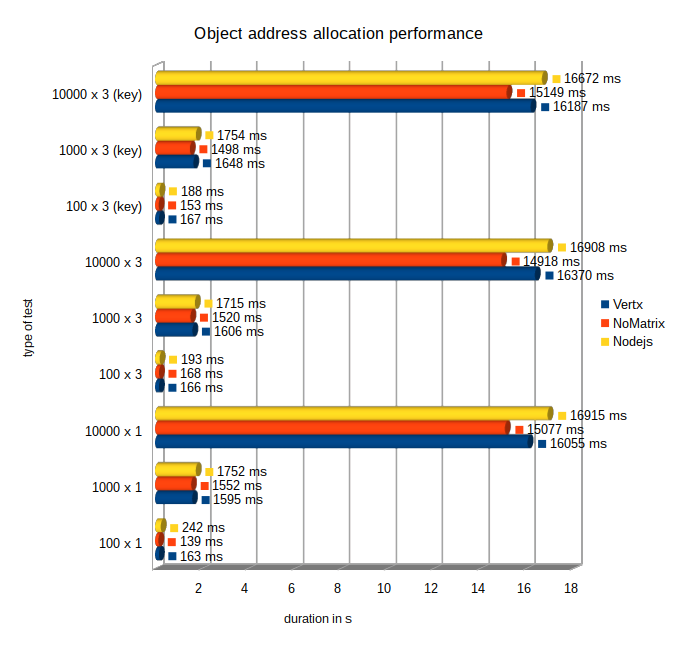


Figure 4: Performance of Object address allocations

All 3 nodes passed successfully all object address allocation tests. The NoMatrix MN performs slightly faster than the Vertx, which again is slightly faster than the Nodejs MNs. But in general the performance is on comparable level.

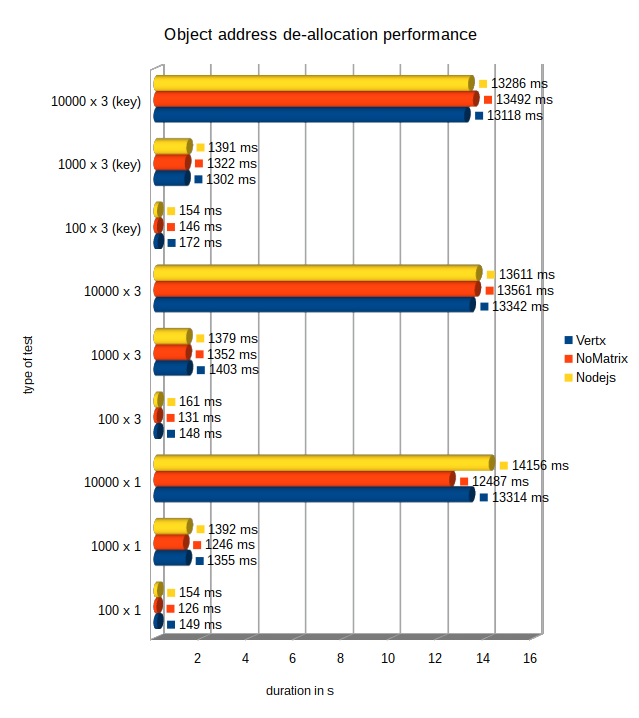


Figure 5: Performance of Object address de-allocations

All 3 nodes passed successfully all object address de-allocation tests. Also for the object addresses the performance of the de-allocations is in general up to 20% better than for the allocations.

The performance of the 3 nodes is on a comparable level, again with a small advantage for the NoMatrix MN in most of the tests.

***Hyperty messaging performance***

The following figure summarizes the results of the performance for the sending of message with different sizes from one Hyperty to another.

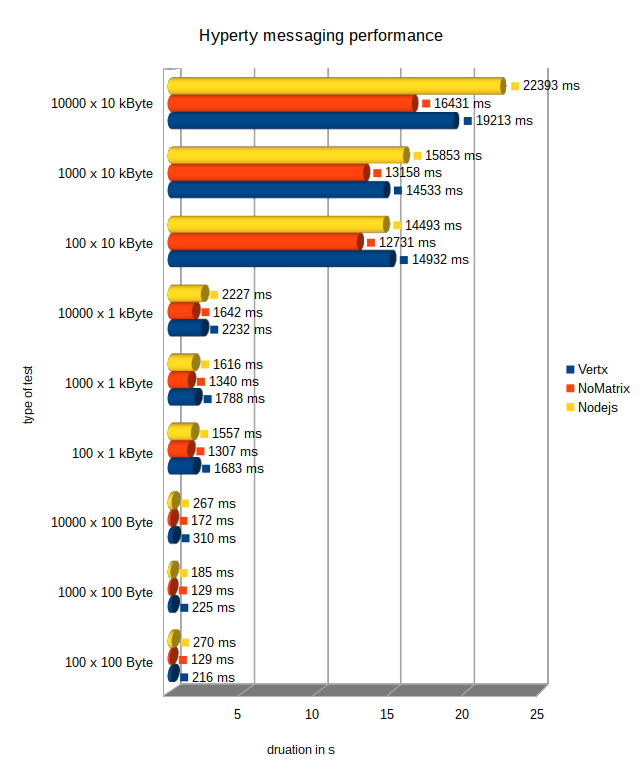


Figure 6: Performance of Hyperty messaging

All 3 nodes have passed successfully all hyperty messaging performance tests.

The NoMatrix MN shows the best performance in all different runs while the NodeJS and Vertx node are on nearly the same level.

***Object status update performance***

The next figure shows the results of the performance measurements for the delivery of object update messages to a variable number of subscribers.

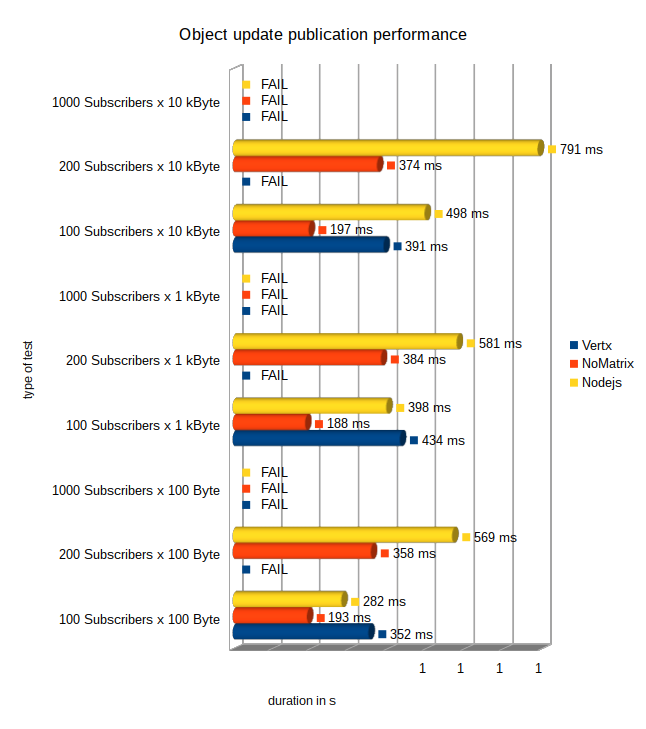


Figure 7: Performance for object update messages to N subscribers

**Note:**

All tests with 1000 subscribers for a event were expected to fail for all 3 nodes. The reason is that the tests were performed using the karma testscripts inside of a google-chrome browser. Since the common browsers have resource limitations regarding the parallel established of outgoing connections, the tests with 1000 subscribers were inhibited. The empirical limit for parallel connections was found around 250.

The NoMatrix and the NodeJS MNs have passed all tests with 100 and 200 subscribers and, as expected, failed for 1000 parallel subscribers due to the mentioned browser limitations. The NoMatrix node performed conspicuously faster than the NodeJS node.

The Vertx node has passed the tests with 100 but not with 200 subscribers. The reason for the failure was that not all subscribers received an event notification when an object update was performed. Variying numbers between 4 and 9 events got lost. Since the tests count only successful when all subscribers receive the update notifications the test failed. The issue has been reported and the reason for these lost events was still under investigation by the component owner at the time of the test documentation.

#### MN Assessment conclusions and recommendations

The tests have shown that the NoMatrix MN can be recommended as the “all purpose” MN, passing all mandatory conformance- and performance- tests and providing the best performance in most of the cases.

The Vertx MN is passing all conformance- and most of the performance- tests. It is the standard MN for most of the applied testbeds in reTHINK and has proven its reliability in the daily work. The lost events for very high numbers of subscribers for updates on a data object seem to have a rather low relevance in the real life operation, according to the practical experiences. Nevertheless they indicate a potential problem in the routing engine that must be investigated.

The NodeJS MN is showing very good performance results but it is failing in the tests for the domain-registry interaction, which are crucial for the use of the MN in a testbed. These failures can be fixed by the proper integration of the latest RegistryConnector version. If this is done the NodeJS MN is a good option for the operation of a reTHINK testbed.