A Framework for Testing AIS Implementations

Tamás Horváth and Tibor Sulyán

Dept. of Control Engineering and Information Technology, Budapest University of Technology and Economics, Budapest, Hungary
{tom, stibi}@iit.bme.hu

Abstract. Service availability has become one of the most crucial parameter of telecommunications infrastructure and other IT applications. Service Availability Forum (SAF) is a leading organization in publishing open specifications for Highly Available (HA) systems. Its Application Interface Specification (AIS) is a widely accepted standard for application developers. Conformance to the standard is one of the most important quality metrics of AIS implementations. However, implementers of the standard usually perform testing on proprietary test suites, which makes difficult to compare the quality of various AIS middleware. This paper presents a testing environment which can be used to perform both conformance and functional tests on AIS implementations. The results and experiences of testing a particular AIS middleware are also summarized. Finally we show how to integrate our testing environment to be part of a comprehensive TTCN-3 based AIS implementation testing framework.

Keywords: Application Interface Specification (AIS), Conformance Testing, Functional Testing, Service Availability

1 Introduction

Service Availability Forum's Application Interface Specification (SAF AIS) [1] defines a standard for a distributed middleware which can be used to implement highly available carrier-grade services. Several implementations of the specification, both commercial and open-source, are available to developers. To select the most appropriate product, thorough testing is required. One of the most important quality parameter of AIS implementations is standard compliance, but performance characteristics have to be taken into consideration when choosing the appropriate middleware.

Due to the complexity and distributed nature of AIS services, testing of the implementations of the standard requires specialized test systems. Implementers of AIS usually perform functional and performance testing on proprietary environments. However, these systems cannot test other implementations, thus they are unable to perform comparative examinations.

In this paper we present a new test suite which can be used to perform conformance, functional and performance tests on various AIS implementations. First, we give a short summary of the specification and the services it provides. Next,

we evaluate some public test systems designed for AIS implementation testing. In the second part of the paper, the high level design of our proposed framework is introduced, followed by the test experiences and results of a particular AIS middleware product. Finally, we sketch the development direction of the system to become part of a TTCN-3-based testing framework.

2 Application Interface Specification Overview

Application Interface Specification defines Availability Management Framework (AMF) and a set of services offering functionality which supports the development of highly available applications. AMF provides a logical view of the cluster and supports the management of redundant resources and services on it. To build highly available services, AMF functionality is extended by a set of basic services, grouped into service areas:

- *Cluster Membership Service (CLM)* maintains information about the logical cluster and dynamically keeps track of the cluster membership status as nodes join or leave. CLM can be notify the application process when the status changes.
- *Event Service (EVT)* offers a publish-subscribe communication mechanism based on event channels which provide multipoint-to-multipoint event delivery.
- *Message Service (MSG)* is a reliable messaging infrastructure based on message queues. MSG service enables multipoint-to-point communication.
- *Checkpoint Service (CKPT)* supports the creation of distributed checkpoints and the incremental recording of checkpoint data. Application failure impact can be minimized by resuming to a state recorded before the failure.
- *Lock service (LCK)* provides lock entities in the cluster to synchronize access to shared resources.

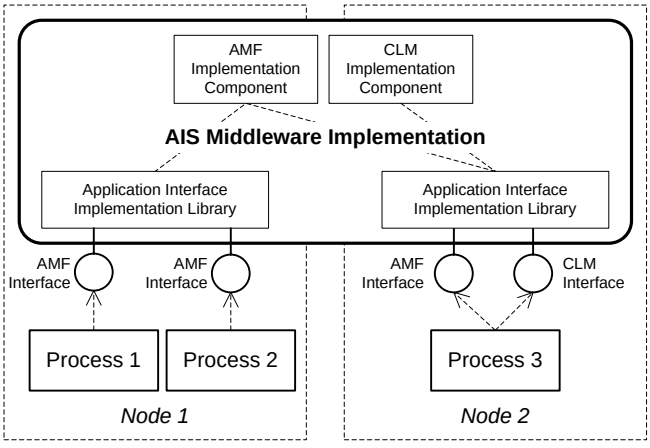

Fig. 1. Interaction between the AIS middleware implementation and the application processes on a two-node cluster.

The detailed description of AIS services is out of the scope of this paper, an in-depth overview can be found in [1]. Nevertheless, the interfaces defined by the specification needs to be discussed here, because these interfaces can be considered as the only points of control and observation (PCO) of the AIS implementation. The relation of the middleware and its clients is shown on Figure 1.

Services provided by the middleware implementation are used by application *processes*. The term *process* can be considered equivalent to that defined in the POSIX standard. Communication between the application processes and AIS implementation is managed by *Application Interface Implementation Library (AIIL)*. Interfaces and implementations of the service areas are separated. Moreover, *Implementation Components* are not covered by the standard; the internal design of the middleware is unknown to the middleware tester. Service area interfaces (*AMF and CLM interface*) represent a logical communication channel between *process*es and the AIS implementation. The logical nature of interface connections is emphasized on Figure 2 by displaying two *AMF interface* objects. The standard provides both synchronous and asynchronous programming models for the communication. Moreover, certain requests can be performed either ways.

In general the synchronous API is much easier to use. Synchronous communication is based on blocking API function calls. The user invokes a request by calling an API function. The request is considered performed by the time the function has been returned. Data exchange between the application process and the AIS middleware is realized by the parameters and the return value of the API function. Although the synchronous model greatly simplifies the programming tasks, certain services cannot be used this way. For example, cluster membership change notifications require a mechanism that permits the middleware to send information to the application asynchronously. Long-running requests are another example where synchronous requests are not recommended.

To support asynchronous communication, the standard employs a callback mechanism. The request API function returns immediately to the caller, and a standard-defined *callback function* is called when the request has been completed. Callback functions are standard-defined, but implemented by the process. Since the middleware cannot invoke a function directly in the application process, a notification is sent first on a *selection object* by the AIIL. In response, the process invokes an area-specific dispatcher function which finally invokes the appropriate callback function of the application process. The body of callback functions usually concludes in a response call carrying status information to the middleware. An illustration of a typical asynchronous communication scenario is presented on Figure 2.

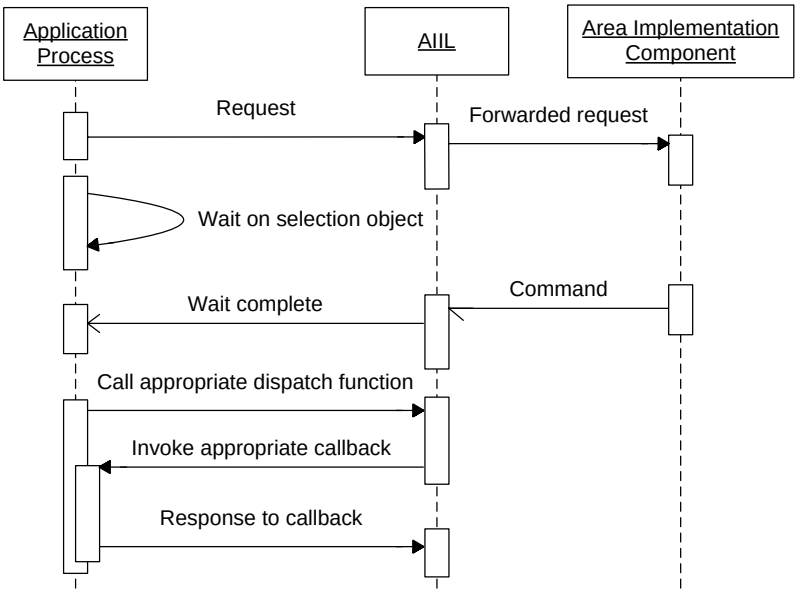

Fig. 2. Sequence diagram of a typical asynchronous communication scenario between the AIS implementation and the application process.

This communication model has high importance considering AIS implementation testing, because all kinds of control and observation tasks can be derived to a series of the following extensions to the model:

- Addition of control operations, such as AIS service requests;
- Inspection of callback function parameters;
- Addition of administrative code (result evaluation, logging, communication with other test components).

3 Current AIS Implementation Testing Systems

Our research covered a survey of currently available AIS testing frameworks. We examined two open source systems considering the following quality parameters:

- **Executable test types.** The most natural expectation from a test system is that it should support multiple test types. We distinguish four classes of tests when testing AIS implementations. Conformance tests address the verification of the API implemented in the middleware product. Functional tests verify that the behavior of the middleware conforms to the standard. Performance tests mean any performance measurements. Robustness tests examine the operation of the system under extreme conditions, for example operation system crashes or hardware

failures. In our research, we primarily focus on conformance, functional and performance tests.

- **Capability of automated testing.** This criterion is also a common expectation from a testing tool. Automated testing means not only automatic test case execution, but also automatic evaluation of results and run-time reconfiguration of the IUT. AIS doesn't define the configuration methods of the cluster, so it can be different in various middleware products.
- **Availability of test results.** The test system should report not only the test verdict, but also additional information about the test case execution. Performance tests for example may require timing information. Additional information is also needed to determine the cause of a particular test execution failure.
- **Adaptability**. This is a requirement specific to AIS standard. The standard evolves constantly, and has multiple versions. Different products realize different versions; even the same middleware may implement different versions of particular service areas. A universal test framework must support this diversity to be able to test multiple implementations.

Based on the criteria above, we shortly describe and evaluate the test suites examined.

3.1 SAFtest

SAFtest [2] is a test execution framework for running AIS and HPI (Hardware Platform Interface) conformance tests. SAFtest test cases are small C programs that implement the procedure shown on Figure 2. In addition, this sequence is extended by calls of AIS API functions with correct and incorrect parameters and execution order. The main purpose of the test cases is to verify that particular API functions exist and yield the expected result when called with different parameters. The framework itself is a collection of makefiles and shell scripts which can configure the IUT, run test cases and collect results. Test cases can be extended with meta-information such as test case description or reference to the specification fragment tested. This meta-information is used by SAFtest to create test coverage reports automatically.

Example test case metadata containing the name of the function under test, assertion description and specification coverage information.

```
<assertions spec="AIS-B.01.01"
 function="saClmSelectionObjectGet">
   <assertion id="1-1" line="P21-38: P22-1">
      Call saClmSelectionObjectGet(), before
      saClmFinalize() is invoked, check if the returned
      selection object is valid.
   </assertion>
   <assertion id="1-2" line="P21-38: P22-1">
      Call saClmSelectionObjectGet(), then invoke
      saClmFinalize(), check if the returned selection
      object is invalid.
   </assertion>
</assertions>
```

This snippet also shows that SAFtest is primarily designed for AIS API function testing as assertions are grouped by the AIS function under test. The range of executable test cases is limited to API conformance tests. The most important flaw of this framework is that test cases of SAFtest ignore the distributed nature of the middleware. Test cases run on a single computer, not on a cluster. This way the majority of the AIS functionality cannot be tested properly.

To summarize, SAFtest is a compact test framework well suited for testing API conformance. It can run tests and generate test result summary automatically. Automatic configuration is not necessary in this case since tests run on a single host. However, this means that most of the AIS functionality cannot be tested with this framework. In addition, testing a different specification version requires a different test case set.

3.2 SAFtest Next Generation

SAFtest-NG [3] is a recent test suite which tries to eliminate most of the limitations of the SAFtest system. Figure 3 shows the main system components of the test suite. The main objectives of this framework are the following:

- To offer a general-purpose AIS test framework which supports not only API conformance but functional and performance tests as well.
- To support fully automatic test execution including automated test environment configuration.
- To be able to test multiple AIS implementations with a minimum amount of reconfiguration overhead.

Test cases in SAFtest-NG are written in Ruby, a high-level object-oriented script-like language. The abstraction level Ruby enables very clean and straightforward test case implementation. Test cases do not run directly on the AIS middleware, they control *drivers*. Drivers are the main components of the SAFtest-NG architecture. A driver consists of three parts. *Driver clients* or short-lived drivers are accepting high-level test case instructions, converting and relaying them to *Driver daemons*. By using this indirection, a single test case is able to drive the whole cluster which AIS implementation manages. *Driver daemons* or long-lived drivers communicate with the AIS middleware via one or more service area interface. Driver daemons implement the communication process shown on Figure 2, and execute AIS function calls. The actual API calling is implemented in separate shared libraries (*clm_driver*, *lck_driver*). This decomposition enables the testing of special implementations, where the service areas are realized according to different versions of the standard. Moreover, this design enhances the reusability of driver libraries when adapting to a new version of the specification.

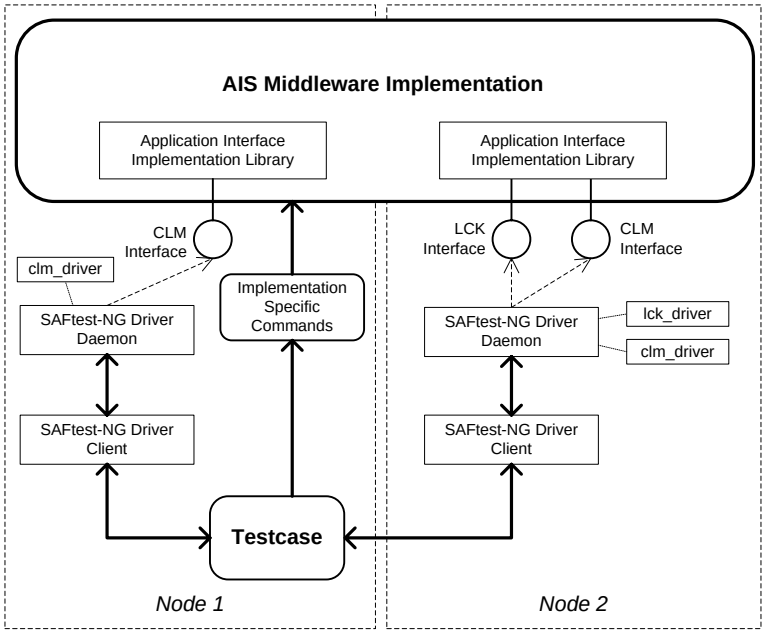

Fig. 3. A sample SAFtest-NG test suite configuration on a two-node cluster. It illustrates relationship between the test case and the various driver components.

SAFtest-NG offers a solution to test environment setup as well by using implementation hooks. These hooks describe the *implementation specific commands* for each particular AIS middleware implementation to perform any operation which is not defined by the standard. These operations include cluster management (addition or removal of nodes), and information retrieval commands (for example, gathering information about the current cluster membership status). AIS middleware vendors only need to provide these hooks to test their product.

SAFtest-NG enhances the executable test range with functional and performance tests. It also supports automated IUT configuration, which SAFtest supported only partially. Support for test result collection is only partial, so test evaluation (especially in case of performance tests) requires log analyzer tools. Unfortunately, SAFtest-NG is an incomplete system, and it seems to be an abandoned project. As Figure 3 suggests, driver libraries are available only for the CLM and LCK service areas, and only for the specification version B.01.01. Consequently, SAFtest-NG cannot be used for a complete in-depth test of AIS middleware.

4 The Message-based AIS Testing Framework (MATF)

In this chapter we introduce the test system (Message-based AIS Testing Framework - MATF) we developed to examine AIS implementations. Our primary design goals were to meet the requirements we enumerated in chapter 3. In addition, future

integration of the system into a TTCN-3 based framework was also an important design consideration. The architectural components of the framework are shown on Figure 4.

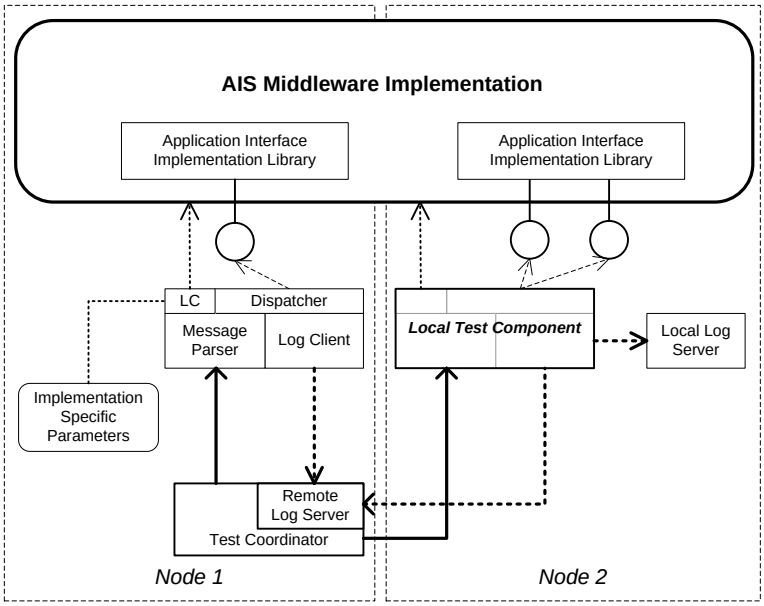

Fig. 4. Components of the proposed Message-based AIS Testing Framework. *Local Test Component* consists of four modules: the *Local Controller (LC)*, the *Dispatcher*, the *Message Parser* and the *Log Client*.

The architecture enables remote testing of the AIS middleware as defined in ISO 9646 [4]. The main idea of MATF is to convert the procedural AIS API into a message-based interface. *Test Coordinator* (*TC)* sends test control messages to the *Local Test Components (LTC)*. LTC then interprets the message with *Message Parser* and either communicates with the middleware (via *Dispatcher*) or controls the cluster node itself (via *LC*). Middleware responses are forwarded to the *Remote Log Server* and potentially to other log servers. Test Coordinator evaluates the results based on the entries of the remote log. In the following chapters we summarize the roles of the components of MATF.

4.1 Test Coordination

Test case messages are sent by the Test Coordinator component. Messages can be transmitted through any reliable communications channel, for example TCP/IP. The TC component implements a log server, which collects incoming data from all Local Test Components. Test case verdict is evaluated based on this data. The format of messages is analogous to function signatures. A message has an identifier and zero or more parameters. This allows the direct mapping of the AIS functions into messages.

Although the use of messages introduces an indirection between the tester and the implementation under test, message-based testing has several advantages over the direct use of AIS API.

The most important among them is the capability of abstraction. Common tasks which require multiple AIS function calls can be encoded in a single message. These tasks include for example connection initiation between the application process and the middleware. Another aspect of abstraction is detail hiding. Messages can hide details of API functions by using default message parameters analogous to C++ default function parameters. When adapting to a new version of the specification, only incremental changes are needed to be performed on the previous Message Parser module. This incremental nature applies also to the test cases. The format of a specific message can be the same for different versions of the specification. Consider two versions of the message queue opening API call [5] [6]:

```
SaErrorT saMsgQueueOpen(
    const SaMsgHandleT *msgHandle,
    const SaNameT *queueName,
    const SaMsgQueueCreationAttributesT
          *creationAttributes,
    SaMsgQueueOpenFlagsT openFlags,
    SaMsgQueueHandleT *queueHandle,
    SaTimeT timeout
);
```

```
SaAisErrorT saMsgQueueOpen(
    SaMsgHandleT msgHandle,
    const SaNameT *queueName,
    const SaMsgQueueCreationAttributesT
          *creationAttributes,
    SaMsgQueueOpenFlagsT openFlags,
    SaTimeT timeout,
    SaMsgQueueHandleT *queueHandle
);
```

Fig. 5. The same API functions from version A.01.01 (left) and version B.01.01 (right) of the specification.

These function specifications have three differences. The return type has changed, the passing form of the *msgHandle* parameter has been altered, and the two last parameters have been swapped. These changes can be hidden from the test case developer. Different versions of the Message Parser modules may translate them to the appropriate AIS function calls.

Another important advantage of the message-based testing is that messages can transparently extend test control instructions by operations that are not covered by the standard, but are necessary to perform successful testing. For example the details of adding a new node to a cluster or completely shutting down a node are not defined in the AIS standard. Specific messages can be implemented in MATF to these operations.

4.2 Message Processing

Messages sent by the Test Coordinator are processed by the Message Parser. MP interprets messages and forwards them to the appropriate communication module (Local Controller or Dispatcher). The Message Parser is an object-oriented recursive-descent parser, which provides high reusability of the parser components, since parsing of different message entities are encapsulated in different parser objects.

Messages specific to local node control are not translated to API functions; rather they are passed to the Local Controller component. The LC will execute operating

system commands to control the cluster or the middleware implementation itself. The concrete effect of control messages can be configured by *Implementation Specific Parameters*, a configuration mechanism similar to *implementation hooks* in SAFtest-NG. This way the test suite can be adapted to test multiple IUTs.

4.3 Controlling and Observing the IUT

Messages that drive the AIS implementation are handled by the Dispatcher component. The Dispatcher performs two main tasks.

Primarily the component provides synchronous and asynchronous interfaces for the Message Parser to enable communication with the AIS middleware. This is implemented by running a *dispatch loop*, which is a generalized version of the communication sequence shown on Figure 2. By default, all requests run on a single thread. However, to test the multi-threaded operation of the middleware,

Dispatcher also has to maintain all session information required to the communication. Session information includes handles, identifiers, or any specification-defined object that persists between multiple API calls. For example, message handle, message queue name and handle parameters on Figure 5 are session information.

After a synchronous operation, the results of the request are immediately available, so dispatcher can forward the results to the *Log Client*. Logging of the results of asynchronous operations is performed in the callback functions. According to its configuration, the Log Client sends the messages to multiple *Log Servers*. A Log Server can be a local file or a process, either local or remote, which collects log data and maintains correct order between log entries. The *Remote Log Server* collects all incoming information from all local test components. Overall test results can be evaluated based on the data collected by the *Remote Log Server*.

5 Testing Experiments

To verify the operability of the architecture above, we executed a set of test cases on OpenAIS [7], an open-source AIS middleware implementation. This chapter summarizes the test results and the experiments we gained during the testing process.

5.1 Test Suite Configuration

The structure of OpenAIS is a straightforward mapping of the standard. Each service area is implemented in a separate process, interconnected by a private communication protocol. The Application Interface Implementation Library (see Figure 1) is implemented by a process called *AIS executive* or *aisexec*. The middleware can be configured by two configuration files, *openais.conf* and *amf.conf*. The former contains operational settings such as network setting or node authorization information. Since this data is implementation-specific, we don't need to alter its contents during testing. As the name suggests, *amf.conf* is used by the Availability Management Framework.

The file stores the actual redundancy model of the cluster. To configure OpenAIS, the behavior of Control Component of MATF has been defined as:

- Adding or removing a node is equivalent to starting or shutting down *aisexec* on that particular node;
- AMF Redundancy model setting is equivalent to the replacement of *amf.conf* on all nodes, followed by a cluster restart. The latter step is needed because OpenAIS doesn't support dynamic redundancy model modification.

To examine OpenAIS, we set up a test suite consisting of three cluster nodes. *Test Coordinator* and *Remote Log Server* relied on a separate controller host. Clocks on all nodes were synchronized from a Network Time Protocol server.

5.2 Test Execution

We have tested OpenAIS version 0.70.1, the latest production ready version available at the time. This release implements version A.01.01 of the Availability Management Framework, and version B.01.01 of the CLM, EVT and CKPT service areas. Distributed Locks (LCK) and Message Service (MSG) are not implemented at all. To test OpenAIS, we established 8 possible configurations of the cluster. The configuration included the number of clusters, the number of local test components and middleware configuration, such as AMF redundancy model.

A total number of 113 test cases had been elaborated based on the specification versions OpenAIS implements. Although the test cases don't provide an exhaustive evaluation of the IUT, they inspect all functionality of the service areas implemented.

Table 1. Summary of the test results.

Total number of test cases	113
Test cases passed	53
Test cases passed but conformance issues encountered	6
Functionality not implemented	31
Test cases failed	21
Test verdict cannot be determined	2

Passed test cases row requires no explanation. *Passed with conformance issue* means that although the functionality under test is correctly implemented, some output function parameters or return values were unexpected. Failed test cases include incorrectly implemented functionality and critical errors of test case execution. Incorrect functionality manifested in invalid data or missing callback invocations. Critical error means unexpected *aisexec* termination which is equivalent to node shutdown. Finally, in two cases the information gathered after the test case execution were insufficient to evaluate the result.

6 Future Work

The test system we developed is far from being a complete framework. We implemented only a prototype version of the architectural elements described above. This prototype system is not capable to perform automated test execution, because of the rudimentary Test Controller and the Log components. The actual purpose of these components is to provide a primitive front-end to the Message Parser and Dispatcher components and to perform actual testing with it.

Nevertheless, the prototypical implementation of the front-end is intentional. The next step of our development is the design and implementation of a TTCN-3 [8] based front end which will replace the Test Controller and the logging components. The new components of the framework are shown on Figure 6.

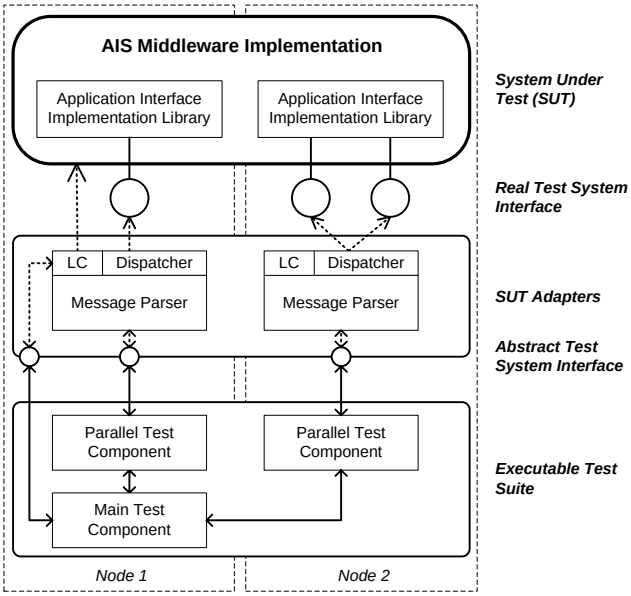

Fig. 6. Integration of MATF components into a TTCN-3 based environment. *Circles* denote TTCN-3 test ports.

The rightmost column of the figure denotes the corresponding elements of the TTCN-3 runtime system architecture [9]. TTCN-3 provides highly sophisticated test coordination and evaluation mechanisms. The *Executable Test Suite (ETS)* can be considered as an advanced replacement of the prototype front-end described in the previous chapter. ETS supports the automation of test execution, test result collection and evaluation. The main modules of MATF (LC, Dispatcher and Message Parser) can be integrated with minor modification into this architecture as *SUT adapters*. *Abstract Test System Interface* is the interface defined by Message Parser and Local Controller. Although this interface already exists, TTCN-3 requires its adaptation to function as TTCN-3 test ports. By the introduction of test ports, the test configuration messages and the actual test messages can be separated. The middleware

configuration messages are sent through a configuration port, while the test messages are sent to Message Parser via a separate port. This is possible because *Main Test Component* not only coordinates *Parallel Test Components*, but can directly send messages to SUT adapters via the Abstract Test System Interface.

7 Conclusion

In this paper we examined two currently available AIS implementation testing frameworks. We found that both systems can be used for particular testing tasks. Nevertheless both systems have certain flaws that prevent them from being general purpose test frameworks. We presented the architecture of a new framework which can be used for comprehensive testing of AIS middleware. To test the usability of the new system we implemented a prototype of the framework and a set of functional test cases. We executed these tests on an open source AIS implementation and summarized the results. The success of the testing process showed that MATF can be used to test AIS middleware. The next step of development is the TTCN-3 integration of the framework. We presented the architectural design of the future test system which highly reuses the actual components of MATF.

References

1. Service Availability Forum, "Application Interface Specification, Volume 1: Overview and Models", SAI-AIS-B.01.01.
2. SAFTest. *(http://www.saf-test.org/)*
3. SAFTest Next Generation. *(http://saftest.berlios.de/)*
4. International Organization for Standardization, "Information technology – Open Systems Interconnection – Conformance testing methodology and framework – Part 1: General concepts", ISO/IEC 9646-1:1994.
5. Service Availability Forum, "Application Interface Specification", SAI-AIS-A.01.01.
6. Service Availability Forum, "Application Interface Specification, Volume 6: Message Service" SAI-AIS-MSG-B.01.01.
7. OpenAIS: Standards-Based Cluster Framework. *(http://developer.osdl.org/dev/openais/)*
8. European Telecommunications Standards Institute, "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language", ETSI ES 201 873-1 (v3.1.1), Sophia Antipolis, June 2005.
9. European Telecommunications Standards Institute, "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 5: TTCN-3 Runtime Interface", ETSI ES 201 873-1 (v3.1.1), Sophia Antipolis, June 2005.