The TENA Middleware

Supporting Real-Time Application Development
for the
DoD Range Community

J. Russell Noseworthy
TENA Software Development Lead
j.russell.noseworthy@objectsciences.com

Tuesday, July 15, 2003
Abstract

The TENA Middleware supports distributed real-time application development. These applications appear in many domains, such as telecom, aerospace, military testing and training ranges, and financial services. The TENA Middleware combines the concepts of CORBA distributed objects and anonymous publish-subscribe data dissemination to provide a programming abstraction known as a stateful (as in “full of state or data”) distributed object (SDO).

Every SDO can have both a remote method interface and publication state. An SDO’s publication state is disseminated to applications based on subscription characteristics, such as the type of the SDO. Subscribers can read the publication state of an SDO as if it were a local object. The SDOs provided by the TENA Middleware metaobject model support inheritance from other SDOs, containment of other SDOs, and references to other SDOs.

Reliability is frequently the most critical aspect of a real-time application, even more important than raw speed. Ultimately, reliability is determined through the course time, after considerable testing. However, the time required to achieve the desired reliability can be greatly reduced with the help of middleware and tools to reduce the possibility of human error.

The TENA Middleware uses UML-based model-driven automated code generation to reduce the amount of software that must be written (and tested) by humans. Furthermore, the TENA Middleware provides the application developer with a powerful distributed shared memory programming abstraction. This programming abstraction is easy for the application developer to understand, resulting in applications with fewer mistakes.

The TENA Middleware API relies heavily on compile-time type-safety to help ensure reliable behavior at run-time. Careful API design allows a great number of potential errors to be detected at compile-time that might otherwise go unnoticed until run-time---where the cost of an error could be extremely high.

The implementation of the TENA Middleware uses C++, as well as a real-time CORBA ORB. The TENA Middleware is presently in use at dozens of DoD testing and training ranges across the country. The number of ranges developing applications with the TENA Middleware grows steadily.
Foundation Initiative (FI) 2010 is a joint interoperability initiative of the Director, Operational Test and Evaluation (DOT&E) of the Office of the Secretary of Defense. The vision of FI 2010 is to enable interoperability among ranges, facilities and simulations in a quick and cost-efficient manner, and to foster reuse of range assets and future range system developments.

TENA Introduction

- Design and prototype a technological infrastructure to enable interoperability and reuse within the range community
  - Seamless environments that integrate test ranges and facilities, training ranges, laboratories, and modeling and simulation (M&S) assets
  - Improve the scope and scale of testing and training the increasingly complex systems and missions in a cost-effective manner

- Recognize that our solutions need to be more than quality software, we need to:
  - Elegantly solve key usability issues
  - Satisfy the core operational and performance requirements
  - Work with the range community so the solutions are implemented

- Lay the groundwork for full support for integrated multi-range events
Operational Driving Requirements

A. TENA must support the implementation of logical ranges, including the management of both software and data throughout the entire event lifecycle, including the planning, execution, and analysis phases.

B. TENA must support the Joint Vision 2010/2020 by providing the foundation for testing and training in a network-centric warfare environment.

C. TENA must support rapid application and logical range development, testing, and deployment in a cost-effective manner.

D. TENA must support easy integration with modeling and simulation to advance the DoD’s Simulation-Based Acquisition and Joint Distributed Engineering Plant concepts.

E. TENA must be gradually deployable and interact with non-TENA systems without interrupting current range operations.

F. TENA must support a wide variety of common range systems by meeting their operational performance requirements, including sensors, displays, control, safety, environment, data processing, communications, telemetry, analysis, and archiving.
Interoperability

The characteristic of a suite of independently-developed components, applications, or systems that implies that they can work together, as part of some business process, to achieve the goals defined by a user or users.

Reusability

The characteristic of a given component, application, or system that implies that it can be used in arrangements, configurations, or in system-of-systems beyond those for which it was originally designed.

Composability

The ability to rapidly assemble, initialize, test, and execute a system from members of a pool of reusable, interoperable elements. Composability can occur at any scale — reusable components can be combined to create an application, reusable applications can be combined to create a system, and reusable systems can be combined to create a system-of-systems.
Achieving Interoperability, Reusability, and Composability

**Interoperability** requires:
- A common architecture ➔ TENA
- An ability to meaningfully communicate
  - A common language ➔ TENA Object Model (OM)
  - A common communication mechanism ➔ TENA Middleware
  - A physical connection between the two systems ➔ Network, shared memory
- A common context
  - A common understanding of the environment ➔ TENA Object Model (Environment)
  - A common understanding of time ➔ TENA OM, TENA Middleware
  - A common technical process ➔ TENA Technical Process

**Reusability** and **Composability** require the above, plus
- Well defined interfaces and functionality for the application to be reused ➔ Reusable Tools, Repository
The TENA Middleware

The TENA Middleware is designed to enable the rapid development of distributed applications that exchange data using the publish-subscribe paradigm. While many publish-subscribe systems exist, few possess the high-level programming abstractions presented by the TENA Middleware.

The TENA Middleware provides these high-level abstractions by using auto-code-generation to create a complex real-time CORBA application. In so doing, the TENA Middleware offers programming abstractions not present in CORBA and provides a strongly-typed API that is much less error-prone than the existing CORBA API. These higher-level programming abstractions combined with an API designed to reduce programming errors enable users to quickly and correctly express the concepts of their applications. Re-usable standardized object interfaces and implementations further simplify the application development process.
An SDO is an abstract concept formed by the combination of a CORBA distributed object with “publication state”. The publication state are data attributes of the SDO that are disseminated via publish-subscribe. CORBA middleware provides the illusion that a distributed object's methods exist on a local object in the native programming language of the application. Unbeknownst to the application programmer, the distributed object's methods may in fact involve a remote invocation to another application on the network. An SDO extends this concept to include, not only methods, but data attributes as well.

The TENA middleware provides the illusion that an SDO's methods and state can be accessed as if the SDO was an object native programming language of the application. Thus, the TENA Middleware offers a distributed object-oriented shared memory programming abstraction to the application programmer, thereby greatly simplifying their development experience.

Every SDO can singly inherit from another SDO or can multiply inherit from an interface. The data (or state) making up an SDO can consist of the typical fundamental data types, vectors, local classes and messages, other SDOs and pointers to other SDOs.
The TENA Middleware provides a high-level, distributed shared memory programming abstraction to the developer. The foundation of the TENA Middleware is the abstract concept of a “Stateful Distributed Object” (SDO). An SDO is a combination of two powerful concepts:

- a distributed object paradigm (like the one used in CORBA)
- a distributed publish and subscribe paradigm.

Benefits of this combination:

- A conventional distributed object-oriented system offers no direct support to the user for disseminating data from a single source to multiple destinations.
- A conventional publish-subscribe system does not provide the abstraction of objects with a set of methods in their interface.
- Interface to SDOs is a lot simpler and more usable than the combination of interfaces to their underlying technologies.

The TENA Middleware uses advanced model-driven code generation to automatically generate a complex CORBA application for the developer.
The abstract concept of a single SDO instance is implemented using a servant object in a server process and proxy objects in client processes. Method invocations on a proxy are delegated to the servant via RMI. Data attribute accesses on a proxy are performed via a local cache read.
TENA specifies an architecture for applications participating in distributed executions. In the range community, the applications are called “range resources” and the distributed executions are called “logical ranges”.
TENA specifies a peer-to-peer architecture. Applications can be both **clients** and **servers** simultaneously. Server applications contain one or more SDO **servants**. Client applications obtain **proxies** to servants by subscribing. Only the SDO’s servant can modify its publication state.

The TENA Middleware, the TENA objects, and the user’s application code are compiled and linked together.
SDO Summary

- **Object Interface**
  - Location-transparent interface to methods, like a CORBA interface

- **Publication State**
  - Data that is disseminated from servants to proxies
  - Dissemination done via publish/subscribe
  - Publication state can be read as if it were local data
  - Publication state is observable
Local Classes and Messages

A local class in the TENA meta-model is an interface to an object whose methods and data are implemented in the local application, i.e., they are not remote methods and the data is not disseminated. Local classes are analogous to CORBA valuetypes. Local classes greatly improve the TENA Middleware application developer's productivity by allowing for standardized re-use of common objects and methods that can be exchanged between distributed applications. One common use is to exchange a position in a local class that contains methods to convert from one coordinate system to another, thereby greatly enhancing interoperability.

A message is a local class that can be disseminated without being part of the publication state of an SDO. Messages are provided in the TENA meta-model to allow application developers to model events that do not persist. An SDO, is intended to model a concept in the application that persists and whose state gets modified over time. A message is intended to model a one-time event, such as "Launch Missile".

Taken in combination, the TENA meta-model and it's implementation in the TENA Middleware provides a programming experience that is both simpler and more powerful than CORBA alone for a broad variety of distributed applications.
What is a Meta-Model?

A metamodel is “a model that defines an abstract language for expressing other models.” from Common Warehouse Metamodel specification by Dr. Daniel T. Chang

The TENA Metamodel describes the features of objects defined in an Object Model (OM). Object Models are defined using TENA Definition Language (TDL).

Why is it important?

The TENA Metamodel is the architectural construct that specifies both the rules for defining an OM and the requirements for the TENA Middleware.

Is the TENA Metamodel finalized?

While the majority of the metamodel has been finalized, some minor details are still under study and await prototyping. Finalizing the metamodel is a high priority for the FI2010 project.

The TENA meta-model defines the high-level programming abstractions provided by the TENA Middleware. The TENA meta-model presently includes: stateful distributed objects (SDOs), interfaces, local classes, and messages. Data streams will soon be added to the meta-model.
The TENA Metamodel supports composition, i.e., the ability to one SDO to be composed of other SDOs. In the example below, the **Platform** SDO contains in its publication state a **Sensor** SDO with the name `longRangeSensor`.

**UML**

```
Platform
- fuel : float
- bestSource : String
- longRangeSensor : Sensor
```

**TDL**

```
class Sensor {
    string point (in float azimuth, in float altitude, 
inout float power);  
    string state;
    boolean onTrack;
    string trackingMode;
};
```

```
class Platform {
    float fuel;
    string bestSource;
    Sensor longRangeSensor;
};
```
TENA 2002 Metamodel
Why use compiled-in object definitions?
- **Strong type-checking**
  - Don’t wait until runtime to find errors that a compiler could detect
- **Performance**
  - Interpretation of methods/attributes has significant impact
- Ability to easily handle complex object relationships
- Conforms to current best software engineering practices

How do you support compiled-in object definitions?
- Use a language like CORBA IDL to define object interface and object state structure
- Use a compiler to implement the required functionality

Thus the concept of the **TENA Definition Language (TDL)** was created
- Very similar to IDL and C++
- All TENA SDOs are defined in TDL
The Challenges of DRE Systems

The most important design-time tools and techniques needed to support next-generation large-scale distributed real-time and embedded systems are:

- Higher-level programming abstractions
- Auto-code generation
- Bug prevention

The first two items have already been addressed. The last item, “bug prevention” is addressed in the TENA Middleware by careful API design.

Reliability is frequently the most critical aspect of a real-time application, even more important than raw speed. Ultimately, reliability is determined through the course time, after considerable testing. However, the time required to achieve the desired reliability can be greatly reduced with the help of middleware and tools to reduce the possibility of human error.

The TENA Middleware uses UML-based model-driven automated code generation to reduce the amount of software that must be written (and tested) by humans. Furthermore, the TENA Middleware provides the application developer with a powerful distributed shared memory programming abstraction. This programming abstraction is easy for the application developer to understand, resulting in applications with fewer mistakes.
The TENA Middleware API relies heavily on compile-time type-safety to help ensure reliable behavior at run-time. Careful API design allows a great number of potential errors to be detected at compile-time that might otherwise go unnoticed until run-time—where the cost of an error could be extremely high.

Strong typing prevents the accidental misuse of an object, increasing the likelihood that an application that compiles will run successfully. Modern C++ makes it possible to eliminate the simple memory errors that have been the bane of many a developer’s application.

The TENA Middleware API provides clear, consistent, and unambiguous memory semantics. Application developers never use delete on TENA Middleware objects. All dynamically allocated objects manage their own memory. In addition to preventing memory leaks, this provides exception safety.
package OMsample
{
    local class VelocityLTP
    {
        double x;
        double y;
        double z;
    };

    local class Orientation
    {
        double heading;
        double pitch;
        double roll;
    };

    local class SensorTrack
    {
        double targetRange;
        double dataQuality;
        string algorithm;
        VelocityLTP reportedVelocity;
    };

    interface Controllable
    {
        string initialize();
        string start();
        string stop();
    };

    class Duration
    {
        long seconds;
        long nanoSeconds;
    };

    class Time
    : extends Duration
    {
        string epoch;
        string toPrettyString();
    };

    class LocationLTP
    {
        double x;
        double y;
        double z;
        double originLatitude;
        double originLongitude;
        double originElevation;
    };

    class TSPI
    {
        Time timeStamp;
        LocationLTP position;
        VelocityLTP velocity;
        Orientation orientation;
    };

    class Participant
    : implements Controllable
    {
        string name;
        string type;
        long ID;
        long displayColor;
        string iconScheme;
        long trackLength;
        TSPI timeSpacePositionInfo;
    };

    class Sensor
    : extends Participant
    {
        string state;
        boolean onTrack;
        string trackingMode;
        vector<SensorTrack> sensorTracks;

        string point(
            in double azimuth,
            in double altitude,
            inout double power);
    };

    class Platform
    : extends Participant
    {
        double fuel;
        string bestSource;
        Sensor longRangeSensor;
    };
}; // End of OMsample
// Create a LocalMethodsFactory
OMsample::Platform::LocalMethodsFactoryInterfacePtr
pPlatformLocalMethodsFactory(
    new OMSample::Platform::BasicImpl::LocalMethodsFactoryImpl);

// Create a RemoteMethodsFactory
OMSample::Platform::RemoteMethodsFactoryInterfacePtr
pPlatformRemoteMethodsFactory(
    new OMSample::Platform::BasicImpl::RemoteMethodsFactoryImpl(
        pPlatformLocalMethodsFactory));

// Inform the middleware that you will publish Platform SDOs
// Provide it the Local and RemoteMethodsFactory and get
// back the Platform::ServantFactory
OMSample::Platform::ServantFactoryPtr_t pPlatformServantFactory(
    pSession->
    createServantFactory<OMSample::Platform::ServantTraits>(
        pPlatformRemoteMethodsFactory, pPlatformLocalMethodsFactory));

// Instantiate the servant SDO using the ServantFactory
// In this example the servant’s state will be disseminated
// using UDP multicast (BestEffort)
OMSample::Platform::ServantPtr platformServantPtr(
    pPlatformServantFactory->
    createServantUsingDefaultFactory(TENA::Middleware::BestEffort));
To change a servant’s publication state, the TENA Middleware uses the concept of “updaters”

Updaters allow sets of publication state attributes to be modified “atomically” (all at once, all or none)

```cpp
// Get an updater
std::auto_ptr< OMsample::Platform::PublicationStateUpdater > pUpdater(
    platformServantPtr->createPublicationStateUpdater() );

// Update the publication state
pUpdater->setFuel( 20.0 );
pUpdater->setName( "M1A1-001" );

// Commit the changes atomically
platformServantPtr->modifyPublicationState( pUpdater );
```
// Define a collection to hold discovered proxies
std::list< OMsample::Platform::ProxyPtr > discoveredPlatformProxyPtrsList;

// Instantiate a discovery callback factory
OMsample::Platform::DiscoveryCallbackFactoryPtr
pPlatformDiscoveryCallbackFactory( new
    OMsample::Platform::BasicImpl::PlatformDiscoveryCallbackFactory(
        discoveredPlatformProxyPtrsList ) );

// Instantiate a destruction callback factory
OMsample::Platform::DestructionCallbackFactoryPtr
pPlatformDestructionCallbackFactory ( new
    OMsample::Platform::BasicImpl::PlatformDestructionCallbackFactory(
        discoveredPlatformProxyPtrsList ) );

// Instantiate a state change callback factory
OMsample::Platform::StateChangeCallbackFactoryPtr
pDefaultPlatformStateChangeCallbackFactory ( new
    OMsample::Platform::BasicImpl::PlatformStateChangeCallbackFactory );

// Subscribe to Platforms
pSession->subscribe< OMsample::Platform::ProxyTraits >(  
    pPlatformLocalMethodsFactory,  
    pPlatformDiscoveryCallbackFactory,  
    pPlatformDestructionCallbackFactory,  
    pPlatformStateChangeCallbackFactory );
Assume discovery occurred and the proxy was assigned to `platformProxyPtr`.

Using the “get” method to read the publication state:

```cpp
// Get a read-only copy of the publication state
OMsample::Platform::ImmutablePublicationStatePtr
pPlatformPublicationState (platformProxyPtr->getPublicationState());

// read an attribute
std::string platformName (platformPublicationState.getName());
```

Or do it all at once:

```cpp
std::cout << "UserExample::FunctionalityTest:: Platform "
  << platformProxyPtr->getPublicationState()->getName() << " has updated fuel to: "
  << platformProxyPtr->getPublicationState()->getFuel() << std::endl;
```
Acknowledgments

This work is sponsored by the Foundation Initiative 2010 Project Management Office. This presentation describes the work of a team of software professionals, including not just the presenter, Russ Noseworthy, but also the members of TENA Middleware development team: Steve Bachinsky, Jon Franklin, Marvin Greenberg, Mike Mazurek, Sean Paus, Ed Powell, and Ed Skees, as well as our new team member, Braden McDaniel. The entire TENA Middleware development team would like to thank the FI 2010 project team, in particular George Rumford, Jason Lucas, Kurt Lessmann and the OM Team, and Kevin Alix.