Enea Element®:
Simplify Distributed Systems with Frameworks from Enea Element

Joe Kidder
Chief Architect, Element Middleware
CTO Office

Distributed systems range from simple multi-threaded applications to multi-slot chassis-based systems to networked clusters of servers. Topologies get more complex when these systems move into cloud-based environments, and more diverse when they involve machine-to-machine (or M2M) solutions. Providers of distributed system software solutions face a number of challenges in building, debugging and maintaining a set of connected applications. Managing these systems requires powerful modeling and a variety of management interfaces to meet a diverse set of needs. The services provided by a distributed system often require a high level of availability. The middleware frameworks that make up Enea Element address many of these challenges.

Introduction
Distributed software systems can range from simple multi-threaded applications in a shared memory space to multiple processes with protected memory on a single compute node to clusters of multiple nodes. They can extend beyond clusters to both multiple cluster configurations as well as asymmetric machine-to-machine (M2M) solutions where a potentially large number of small, simple devices communicate with a central controller.

Distributed systems come in a range of shapes and sizes. In addition to the traditional chassis with multiple slots connected by a networked control plane, there are multi-core devices with multiple compute domains, networked clusters of rack-mounted servers, and cloud-based clusters. Developing software solutions that run on these types of distributed systems is greatly simplified by taking advantage of various off-the-shelf frameworks. A Messaging Framework simplifies writing connected yet independent applications. A Debug and Trace Framework simplifies the
Messaging between threads is a uniform way of communicating in a distributed system, whether the threads are intra-process, inter-process, inter-node, or inter-cluster. Messaging challenges include the addressing of messages to communication partners, as well as how the messages are transported between process spaces and between physically separated CPU domains.

Software developers of a distributed system encounter complex relationships amongst multiple independent applications, often spanning multiple nodes. After the basic functionality of each application has been debugged and tested, the integration of the complete solution presents a debugging challenge for the integrated functionality. The characterization of the performance and scaling behaviors of the distributed system can also require sophisticated visibility and access.

A common management interface to a distributed system is much simpler than individual management of the various pieces that make up the system. The management entity may be an operator interfacing directly to a simple command line interface (CLI) or a more powerful web-based interface. An operator may also use an Element Management System (EMS) or a Network Management System (NMS) that communicates with the distributed system over a management protocol like SNMP or an XML-based protocol like NETCONF or XML-RPC. It is common to see a variety of management interfaces, to accommodate both direct operator management as well as a machine intermediary (e.g. EMS or NMS).

Many distributed systems require some degree of high availability. This may be as simple as error monitoring in applications and error recovery by application restart in a non-redundant system. Typically, high availability is built around managing redundant nodes for critical functionality. In these latter cases, the redundancy model may be a 2N (i.e. 1:1 or 1+1) active/standby relationship. When the 2N relationship becomes too expensive, an N+1 redundancy model may replace it, where one standby node may back up several active nodes.

Challenges in Building Software for Distributed Systems
Software developers face a number of challenges when designing, implementing, and deploying a distributed system. The nature of a distributed system requires communication between entities in the system. The most basic entities of a software solution are threads of execution. In the simplest case, threads may have access to the same memory space and can access shared data. Threads in a distributed system often do not share access to the same memory; they are often run in the protected memory spaces of different processes or within entirely separate CPU and memory domains on different nodes.

Characteristics of a Useful Messaging Framework
Challenges:
There are many alternatives for solving individual messaging problems. There are high-level messaging solutions that provide a combination of simplifying and/or powerful messaging behaviors. However, they often don't apply to high performance or large scale use cases that don't need the high level behavior. At the other end of the complexity spectrum, Linux provides a number of Inter Process Communication (IPC) services that scale well and have good performance, including pipes, sockets, and System V shared memory and queues. However, these mechanisms can be disjoint and don't provide a common software interface. Further, the high level messaging services don't have a common heritage from the fundamental IPC mechanisms available in Linux. There is no unified progression from scalable high performance IPC to value-rich messaging solutions.

Figure 2: Clusters in Chassis, Rackmount Servers or the Cloud
Messaging between CPU domains or nodes is a major part of any messaging solution. This might be from one CPU core to another within a homogeneous multi-core device or between a general purpose CPU core and one or more DSP cores in a hybrid multi-core device. In both of these cases messaging might take place over a shared memory interface. In most chassis designs, there is either a specialized messaging fabric or an Ethernet control network, often provided by an integrated L2/L3 switch. For distributed systems that span multiple clusters, there is often only L3 or IP connection between clusters. In cloud-based infrastructures, the virtualization of the network may only support IP connectivity between the virtualized nodes.

**Recommendation:**
Software architecture for a distributed system should be built on a scalable unified messaging framework. A useful approach is one that progresses from a fundamental IPC service through a layer of messaging services that extend the IPC to a set of value-rich messaging solutions. The software interfaces at each layer in this messaging framework hierarchy should be consistent with the layers below. An application should be able to use any or all of the layers with a clear understanding of how each is related to the others.

A modular plug-in architecture is recommended to address the range of node interconnect types. This interconnect architecture should scale down to high-performance, hardware-centric mechanisms that may also have restricted resources. Likewise it should scale up to support high functionality interfaces such as Ethernet or IP that provide a greater degree of reach between nodes.

**Solution:**
LINX as Foundation IPC
Enea recommends the use of LINX, an open source offering, as the foundation IPC of a useful messaging framework. LINX uses discovery to find endpoint handles, given an endpoint name and a path (a list of any inter-node connections necessary to reach the endpoint), and uses endpoint handles as destinations for sending messages. The endpoint handles are the same format whether the message is intra-process, inter-process, inter-node, or inter-cluster. Discovery can be dynamic. If an endpoint discovery is requested for an endpoint that either doesn’t yet exist or can’t be reached due to a missing connection, the discovery will resolve at such time as the endpoint exists and is reachable. Discovered endpoints can be supervised, resulting in notification if the endpoint either ceases to exist or becomes unreachable. The results of discovery and supervision requests are delivered to the requester in LINX messages.

LINX is not connection oriented. Each endpoint has a single receive queue for all messages sent from any endpoint. This provides excellent scaling properties. A fully meshed group of N endpoints scales at N rather than at $N^2$ as in a connection oriented solution.

LINX is reliable. For a destination endpoint that is supervised, application software
knows that a message will reach its destination or the application will be notified that the endpoint is no longer reachable.

For connections between nodes, LINX supports a modular, plug-in connection manager architecture. Connection managers can be created for a wide range of media that each provides a consistent behavior. There are a number of connection manager implementations available, including raw Ethernet (L2), shared memory, SRIO (Serialized RapidIO), and TCP. The connection manager architecture provides a reliability layer for unreliable media (like raw Ethernet), but this layer is not used for reliable media (like SRIO or TCP). A LINX messaging solution can include multiple different connection managers.

**Name Server**
The Name Server provides a logical layer of abstraction for service discovery. The LINX discovery mechanism uses a physical addressing scheme for discovery of endpoints. The name of an endpoint and a path describing the inter-node connections to reach the endpoint are provided to a LINX discovery request. There is only one endpoint name for each endpoint, and only one endpoint for a particular endpoint name. The Name Server allows an endpoint to publish a service tag (i.e. a string). Other endpoints that subscribe for the service tag are provided with the endpoint handle for the publisher of the service tag. The subscriber doesn’t need to know the endpoint name or the path to the endpoint to find the service tag publisher – the discovery request is logical rather than physical. An endpoint can publish multiple service tags if it provides multiple services. Multiple endpoints can publish the same service tag, providing subscribers with the endpoints handles of each of the multiple publishers.

Service tags can have scope limited to a node, to a cluster, or to a “super-cluster” – a collection of clusters. The publisher of a service tag can attach additional data, or attributes, to the published tag. When subscribers are notified of the published tag, the attributes are provided to the subscriber.

**Message Dispatch**
A typical message handling loop removes a message from the head of the receive queue and then parses the message to decide how it should be handled. The Message Dispatch Service provides a mechanism by which applications can create a message dispatch registry and then register message signatures (e.g. the message number) and an associated callback routine and context pointer. When a message is received, it is passed to the message dispatch along with the message registry handle, and the registered callback is invoked with the message and the registered context pointer as arguments to the callback. This simplifies the event loop: remove a message from the receive queue and call dispatch with the message. It also allows client libraries for services and message based APIs to populate the dispatch registry with their own message signatures and related callbacks. The client libraries can hide the details of the asynchronous messaging with their respective servers from the client.

**Figure 4: A Scalable Unified Messaging Framework**

- Publish/Subscribe Distribution
- Application-Level Flow Control
- ...and more
- Logical, Service Oriented
- Simplified Message Dispatch
- Discovery and Supervision
- Scalable, High Performance
- Seamless thread/process/node
application, including server discovery and supervision as well as any other messaging activities between client and server. The client application provides its message dispatch registry to the client library initialization function, and the rest of the client-server interaction can be hidden. This simplifies the client code as well as simplifying the evolution of the client server implementation. The private interaction between client and server can change without the public client API changing, allowing upgrades to client and server without impacting the client beyond linking in the new client library.

**Object API**

Message payloads are difficult to pack and unpack when they contain complex data. Variable-sized strings often require complicated string space implementations in the flat representation of a message. Representing optional data and collections of data becomes difficult, and hierarchy can be even more difficult. The possibility of mixed revisions of software running concurrently within a distributed system raises the possibility that messages can be constructed in one version of software and received by another. Care must be taken when adding new items to a message payload. An older application may receive a message and stumble over an unexpected new item from a newer messaging partner, or a newer application may stumble over the absence of an expected item from an older messaging partner. Stumbling when parsing a hard coded message format often results in an application crash.

The Object API is a hierarchical container based mechanism that supports typed data leaves and is designed for building and manipulating structured, self-describing objects. Containers and leaves are named. Data can be safely read from an object. If the data is in the object, it is returned. If the data is not in the object, the return status indicates that the data is not there. The Object API supports the serialization of objects into messages and de-serialization from messages back into objects. In addition, objects can be serialized to XML strings or files and de-serialized from XML strings or files back into objects.

The Object API simplifies the construction of powerful APIs by making it easy to share complex data. It also makes handling syntax changes due to version skew easier and safer.

**Element Messages Framework: Messaging Patterns**

The Event Service is a messaging pattern that is provided with the Element Messaging Framework. The pattern is referred to commonly as a “pub/sub” (publish and subscribe) messaging service. The service distributes messages from publishers, or producers, to subscribers, or consumers. The Event Service creates the concept of a group, to which publishers register as producers and subscribers register as consumers. An event is an encapsulation for a message payload. The encapsulation contains the group id and a type id (to distinguish between event types within a group) and a time stamp of when the event is sent. When a producer sends an event to a group, all of the registered consumers for that group will receive the event. Those who register as producers and/or consumers of a particular group can do so at a node scope or a cluster scope.

The Application Level Flow Control (ALF), is a credit-based flow control mechanism. ALF supports directional groups of a number of senders from one to many that send messages to a single receiver. A pool of credits is distributed by the receiver to the senders. Senders can automatically queue messages locally when they run out of credits and automatically resume and dequeue the messages when credits are replenished. Senders can also get status of “would block” when sending, allowing the sender to implement an appropriate behavior, such as applying flow-control upstream or specific queuing and/or discard behaviors. The sender can be notified through a callback when credits are available.

Other value-rich message services can easily be constructed using a combination of LINX and the Element Messaging Services.

**The Value of a System Wide Debug and Trace Framework**

When building a software system, particularly a distributed system, developers need the ability to look at a variety of connected components in a unified way in order to understand and analyze software and system behavior. After a system is deployed, those who administer, provision, and maintain the system have the same need.

There are two basic types of mechanisms used for analyzing and monitoring system behavior. A logging and tracing mechanism allows one to see historical information, indicating what has been happening in the system in a serial, historical form. An interactive mechanism allows one to inspect current status and state of the system and its components, as well as perform actions on the system and its components.

**Challenges: Log and Trace**

Debugging the system using the management model can only provide insight into the abstract behavior of the system as represented by the model. Internal implementation details of the applications are often not reflected in the management model. In a distributed system, it is valuable to be able to view this data across the whole system in one aggregated view. However, aggregating the data from a number of nodes presents the possibility of data overload, often due to coincident bursts of entries from a number of different nodes. Managing this N-to-1 overloading behavior is an important design point.

Controlling the content of the log, i.e. what log entries are generated, is often done at compile time. This means that developers and system architects need to determine what log data is interesting enough or necessary enough to be generated from the applications and services in the system. However, when debugging a problem, either in the lab or in the field, there are often additional log entries that can provide more detail for the historical data view. This is addressed...
typically by re-compiling special "debug builds" of the software or by using ad hoc methods such as setting flags in applications to generate or not generate log messages.

**Recommendation: Log and Trace**

It is valuable for the log entries to be classified in various ways to make it easier to filter the data when viewing.

A logging service should incorporate a simple, unified mechanism that supports dynamic control of log content through a distributed mechanism that performs filtering at the source, i.e. the applications and services. It should allow full log instrumentation of applications, but enable the generation of the entries to be disabled by default and enabled when needed.

The full life cycle of log information should be considered. Log file size should be managed, and expired log files should be able to be pushed off the system to external storage.

**Challenges: Interactive Debug**

The software in a distributed system typically runs on a number of different CPU domains. There are often multiple layers of abstraction in the software, making the inspection of data through a source level debugger tedious and error prone. The various software modules are typically designed by a number of different developers, sometimes across geographically distributed teams. The software often has real time characteristics. Even if not hard real time, the software may have interdependencies that don't allow intrusive debugging that stops a component for debug at human speed. Lastly the software is often highly available. Similar to real-time constraints, high availability mechanisms are often in place to determine that a lack of response from a software component may indicate an error, resulting in the initiation of error recovery mechanisms.

Debugging complex distributed systems, even multi-program or multi-threaded applications can be very challenging. The typical tools are either source-level debuggers, like gdb, that give stop mode interaction with individual applications and threads or a system-wide embedded management interface.

Debugging individual applications and/or threads using a source-level debugger can be tedious and error prone, particularly when traversing complex data structures. The use of this stop-mode debugging on a running distributed system can also be disruptive, causing other threads and applications that interface with that being debugged to have issues.

**Recommendation: Interactive Debug**

Debugging a distributed system requires insight into the implementation of the software solution. Developers that are

---

**Figure 5: Traditional Interactive Debug Mechanisms**
debugging complex system problems
during integration and during behavior
and performance characterization need
access to implementation specific data
and behaviors. Support and maintenance
staff have similar needs when debugging
issues in the field. The mechanisms
enabling this application level debug
should provide non-disruptive access to
implementation data. Ideally, the data
is provided through instrumentation
put in place by the various component
designers and developers.

In order to support repeatable and usable
access, an interactive debug service should
be easy to use, with a normalized user
interface. Multiple user interfaces support
different debug use-case scenarios. A web
interface is the simplest and easiest
interface to learn, and it provides the
option of the best visualization of the
data. In addition a web interface is
generally less error-prone than other
interfaces. A command line interface (CLI)
requires fewer resources than a web
interface and is also more easily scripted.
A machine-oriented like XML-RPC allows
the integration of test harnesses to a
system or data mining mechanisms for
characterization operations or maintenance
activity. With all of these UI options,
a mediation layer allows one-time
instrumentation of applications.

A central UI connection is recommended,
in order to access software components
spread across a distributed system.

Solution
The Element Debug and Trace Framework
provides services for both historical tracing
and for interactive debug.

Element Log Service
The Element Log Service provides a system
wide logging facility with a centralized
log event repository.

Log Message Classification
The Log Service allows applications to
generate log events that are classified.
The log class and subclass identifies
groups or individual events. A general
type can be applied to any event (the
default set of types is Fault, Configuration,
Accounting, Performance, and Security
- FCAPS). The severity of a log event
indicates importance. The entity ID is a
32-bit number that has relevance within
the application or service that generates
a given log event. Finally the slot label
is a string label for the slot on which the
log event was generated.

The classification attributes are used for
source-level log filtering, which takes
place within the Log Service API within
the client. Log filters can be configured
dynamically at a cluster-wide level, and
the resultant filters are distributed to the
applications and services to which the
filters apply. The classification attributes
are also used for display filtering when
viewing the log through the online and
offline log viewers.

Time Stamps
Each log event is time stamped when
it is generated. The time stamp is an
additional field that can be used by the
display filtering when viewing the log
through the various log viewers. The time
stamps contain a 32-bit seconds field as
well as a 32-bit microsecond field.

Central Persistent Log Database
The log events from throughout the
cluster are aggregated in a single
centralized log server. The log server
maintains a persistent lightweight
“database” of log events that is indexed
by time stamp. There are redundant
log servers in a redundant system. The
log events generated by applications
and services throughout the cluster are
queued on the generating node in a set
of per-severity queues and transmitted
using credit-based flow control to the
central cluster-wide log server (or the
two servers in a redundant configuration).
The flow control ensures that combined
log traffic from a number of nodes
doesn’t overwhelm the central server.
The per-severity queues ensure that lots
of log traffic of low importance doesn’t
result in clipping of higher importance
traffic in cases of extreme overload.

Log File Rollover and Data Push Service
Log files are closed periodically when
they reach configurable thresholds of file
age and/or file size. A data push service
can be configured to transfer closed log
files off the system to up to two different
(a primary and a secondary) off-system
file servers. The log file viewers can view
closed log files that are still on the system.

The Element Command Service
The Element Command Service provides
developers with the ability to instrument
their applications and services with
commands that provide application
specific information. This is done through
a common API that supports a web-based,
CLI-based, and XML-RPC-based user
interface, allowing application developers
to write command code once for all three
user interfaces.

Three User Interfaces
There are three user interfaces to the
Command Service. The most common and
useable interface is the Web Interface.
This provides a full web-based index
into the set of available commands
with support for hierarchical command
grouping as well as different usage
categories. The web interface is the
least error-prone and simplest to use.
In addition to tables (which are also
supported in the CLI), the web interface
can support hyperlinks embedded in
output that issue other commands,
style markup (e.g. red for alert), and
form-based input for providing additional
command arguments.

Dynamic Command Registration
Applications and services within the
cluster register commands by providing
the set of tokens that make up the command
as well as a location identifier for
the command.

Commands are dynamically registered.
When an application or service registers
a command, it is now available to be
issued and will appear in the index frame
and pages on the web or reflected in the
help and command completion in the
CLI. When the application unregisters
a command or if the application is
unloaded, the command is removed
from the set of available commands.

Distributed Command Dispatch
Commands are issued by a user interface
(web, CLI, or XML-RPC) to a cluster-central
Command Server. From that point they are distributed to the application or service that registered the command, based on the tokens that make up the registered command and the location identifier with which the command was registered. The application wakes up in the registered callback, parses any additional arguments for the command, and builds a command response using the Command Service API.

Rich API for Command Response Generation
A variety of features are available to the client application when creating a response in a command callback routine. These include flexible table generation with sort-able columns, formatted for web or CLI, hyperlinks and Alert/OK dialog support.

A Management Framework Simplifies Application Development
A management service is intended to enable a set of applications and services running on a system to be managed by an external management entity, either an operator through a direct user interface or via an Element Management System (EMS) or a Network Management System (NMS). The managed system presents an interface for external management, called an agent. Typical management agent interfaces are Command Line Interface (CLI) or Web, SNMP, NETCONF, and other protocol-based interfaces, which enable an EMS or NMS to communicate with the device on behalf of a GUI-enabled operator.

A model of the information associated with the managed system and its function is necessary to ensure that the management entity has the same expectations as the applications and services in the managed system. This model describes configuration information that is consumed by the managed system, operational data that is provided by the managed system upon request by the management entity, and notifications that are pushed from the managed system to management entities. The information model for a managed system may be informal, perhaps described in a document that is referred to by the developers of the EMS as well as the applications within the managed system. Normally, the model is described using a formal modeling language such as a ASN.1, used in MIB (module information base) definition, or Yang, a modern modeling language developed to complement the NETCONF management interface specification.

Challenges
“Stovepipe” Architecture
There is a common evolutionary architecture in managed system architecture. Often the first version of a device supports a single management interface, perhaps a CLI. The applications and services within the managed system are implemented with direct support for that management interface. As the system matures, additional management interfaces may be required, perhaps SNMP and later an XML or Web interface. Often the support for these additional management interfaces is added directly to the applications and services within the managed system. This leads to what some call a stovepipe architecture, because there are direct pipes leading from the application to each of the supported management interfaces. New applications and services for the system often inherit this architecture, adding complexity and effort to the design and implementation of new features.

Recommendation
Unified Approach: A Mediation Layer
In recent years, a new approach has been created that separates the northbound management agent interfaces from the southbound application interfaces. This is achieved through a mediation layer that presents a single unified southbound interface to the applications and services within the managed device. With this architecture, applications can “write once” the implementation for management access. The northbound interface is typically common enough that support for new management agents can be added. Ideally, the management service supports a plug-in interface for the addition of management interfaces. Thus, a new interface can be added without changing the applications and services in the managed device.

Southbound API
An ideal southbound API for applications includes the following capabilities: configuration updates, operational data requests, actions, and notifications.

The application must be able to receive its configuration as well as updates to its configuration. Ideally, the application can register for only the subset of the model...
in which it is interested. The application must be able to provide operational data when it is requested by a northbound interface. Again, the application should be able to register which parts of the model it owns, including instance specification for model objects. The application must be able to take action, ideally actions that are consistent with the management model. Lastly, the application must be able to provide notification of events and alarm conditions occurring within its domain. These notifications should be related to objects within the management model.

A southbound API may also support validation, which is useful in providing transaction support.

Distributed Southbound API
For a distributed system, the management service should achieve two important goals. To the northbound agent interfaces, the device should look as if it's one single entity. To the client applications, the management service should look as if it's local to the application.

In the first case, the device should look, to the northbound agent interfaces, as if it's one single entity. The northbound management interface should not expose any details of the location or reachability of the underlying applications that implement the managed objects.

In the second case, the application should see the southbound API as always being available and always in a well-known place. In a highly available system, the management service may disappear briefly or even move from one node to another as a result of an error and the subsequent recovery from error. This is not something the application code should have to deal with.

Solution
The Element Management Framework contains services for modeled persistent configuration, operational data, actions, and notification. These services are fully integrated with the Element High Availability Framework and make use of Element Objects for the distribution of management objects across the southbound API. Currently, the framework provides northbound interfaces through which management agents can be integrated such as CLI, Web, SNMP, and XML-based interfaces such as NETCONF or XML-RPC.

The Element Notification Service provides an Alarm Manager for maintaining system-wide aggregate alarm state based on the set of Alarm Notifications that have occurred in the system.

The Capabilities of a World Class High Availability Framework Challenges
As a distributed system grows to include more nodes and more applications, the overall reliability of the system shrinks. Two components of a given reliability, when combined together with any dependency, reduce the reliability of the resulting combination. This increases the likelihood of errors in the system. As the interdependency between components and nodes grows, it becomes unacceptable to incur the loss of service that results from restarting the entire system as a consequence of occurrences that disrupt individual applications or nodes.

The reasons for disruption certainly include the obvious case of failure of a hardware or software component. Additional causes of disruption include administrative actions to hardware or software for maintenance or operational needs as well as the upgrade of software components in the distributed system.

Recommendation
A distributed system that provides highly available services must be designed and implemented with high availability in mind. The architecture of message based interfaces between components in the distributed system should address the dynamic, mobile and elastic nature of messaging partners (both peer-to-peer and client-server). Partners can be dynamic due to restarts; a partner can disappear and reappear. A partner may be mobile due to redundancy; a partner can failover from one active resource to a standby resource. Partners can be elastic for scaling or power management purposes, resulting in the growth or reduction of a set of service providers. The Element Messaging Framework, including LINX, provides a solid messaging architecture on which to design and implement this system architecture. A high availability framework provides additional needed services.

Availability Management Framework
The core of a high availability framework is a framework that manages redundant resources and the assignment of work to those resources to provide service. The framework will detect errors in system components (e.g. applications and nodes), apply recovery of the service provided by the failed components, and support the repair of the failed components. The Service Availability Forum (SA Forum) has created a standard specification for such a framework, called the Availability Management Framework, or AMF. The AMF provides lifecycle management for both components (typically software applications to which errors can be isolated) and nodes. The AMF provides a number of redundancy models with which to manage the assignment of service (i.e. work) to service providers (i.e. workers). These models include 2N, N+M, N-Way, N-Way Active, and None. The 2N redundancy model is a common active-standby model, where one service provider is active and the other standby. This is similar to 1+1 or 1:1 redundancy models, but has the added ability to include spare service providers in case of a lengthy loss of either active or standby. The N+M redundancy model is an extension of a common N+1 redundancy model in which one standby protects multiple active service providers. The goal is that service can be protected by a more economical configuration than 2N provides. The N-Way and N-Way Active redundancy models support work assignments that distribute load across a number of providers. A non-redundant model allows components to be deployed in a non-redundant manner, but still takes advantage of component restart error recovery.
Checkpoint Service
The maintenance of critical state is crucial to provide consistent service throughout the loss of a service provider and subsequent recovery. Care should be taken to limit the amount of critical state that a given application replicates in order to maintain the state. In order to minimize data replication, applications should only replicate data that they own, meaning data that cannot be retrieved from some other application or service within the cluster. It may be data that the application has acquired from an entity external to the system. It may be data that the application has created based on events and exchanges with partners within the system, including decisions that cannot be deterministically recreated or processed versions of data that, while retrievable from other applications, is excessively time-consuming to recreate.

A checkpoint service is useful for replication of an application’s critical data. Data presented to the checkpoint service is held by the service and replicated to a remote node for protection against node failure. In the event of a failure of the application, the critical data can be retrieved from the checkpoint service, whether the failure is recovered by an application restart or by a node failover. It is important that the checkpoint service can support multiple clients, each with one or more sessions of data. The checkpoint service should support the addition, modification, and deletion of data from a checkpoint session. Lastly, it is very helpful if the checkpoint service supports organization of the data within a checkpoint session.

Software Manager
Software management is the last of the major services needed to provide high availability. This service should manage the sets of software images on the system and assist in the upgrade from the current running version of software to a selected target version.

Image management services should include a mechanism to validate the contents of a software release. Validation should check that all of the intended files are in the release and that the files match those intended for the release and haven’t been corrupted during install or after install. If the software release exists on multiple nodes, all valid software releases should be synchronized and replicated to all necessary nodes.

Upgrade capabilities should include support for an in-service upgrade, allowing the system to provide service throughout the upgrade procedure. An in-service upgrade manager may make use of the error recovery mechanisms provided by the availability management framework (e.g. AMF) in order to restart the necessary software applications to move to the new software release. If the applications in the system can provide service through an error and ensuing recovery, that mechanism can be used for upgrade as well. An in-service upgrade should isolate the upgrade actions to affect as few applications and nodes.

Figure 7: AMF Redundancy Models
as possible. Only those applications that have changed between the running release and the target upgrade release should be restarted. The upgrade should be sequenced in order to provide service throughout the upgrade, maintaining critical state along the way.

In the event of a failure of either the new version of software or the upgrade procedure itself, it is imperative that the system is quickly and deterministically returned to running the original software version.

An in-service upgrade is typically a series of application and/or node restarts. This should be an orderly process, and is sometimes referred to as an upgrade campaign. The upgrade campaign can be hand-created for each release combination and for each device configuration. That can be time consuming and error prone. Ideally, a software manager will derive the upgrade sequence from a combination of the difference between the current and target software releases, the configuration and state of the device to be upgraded, and upgrade sequence rules that are consistent with the software architecture of the system.

**Solution**

**Element High Availability Framework**

The Element High Availability Framework provides the three major pieces necessary: an AMF, a Checkpoint service, and a Software Manager.

**Element AMF**

The Element AMF is an implementation of the SA Forum AMF Specification, revision B.02. It has been implemented from the ground up, based on the Element Messaging Framework and integrated with the Element Debug/Trace Framework and the Element Management Framework. The Element AMF is integrated with the Element Software Manager providing application restart, service unit failover, and node failover capabilities to enable an in-service upgrade.

The Element AMF supports dynamic configuration, allowing new services and service providers to be added to a running system. Removal of existing services and service providers is supported as well as modification of some aspects of running service providers, such as dynamic modification of recommended recovery policy for an application. The configuration of the Element AMF model is done through two different mechanisms. The AMF Configuration Wizard provides a fast and simple mechanism for the creation of services and service providers that reduces the chances of error. The AMF Configuration Explorer is a more granular configuration tool that allows more flexibility and access to the complete AMF configuration model.

The Element AMF also supports offline configuration for initial AMF configurations, including web-based support for both the AMF Configuration Wizard and the AMF Configuration Explorer. In addition, AMF configurations can be expressed in a simple scripting language that uses the same interface concepts as the AMF Configuration Wizard.

The Element AMF supports all of the redundancy models in the AMF specification. The Element AMF supports administrative actions that are performed on entities in the AMF model. The administrative actions can be invoked either from the Element Command Service via Web, CLI or XML, or by software in the system via API.

The Element AMF spawns applications, or components, that are specified in the AMF model. Failed applications can be rapidly detected via operating system signals. In addition, an application may report errors in itself or in other applications via the AMF API. Lastly, a configured health check can detect errors due to failed response by an application to periodic health checks.

**The Element Checkpoint Service**

The Element Checkpoint Service provides a mechanism for holding and replicating critical application state. The service supports the concept of a local session that “owns” the data on the same node as the creator of the data. The local session is opened by the active instance of an application. It also supports the concept of a remote session that is opened by the standby instance of an application. The remote instance provides protection against the loss of a node containing both the active application and its local checkpoint server. In the event of an error to an active component that can be recovered via a restart-in-place, the local checkpoint session provides rapid recovery of state by the restarted application. In the event of a failover, the standby can retrieve the state from the previously remote checkpoint session. A standby application can receive live updates from a remote session, allowing the standby to be a hot-standby. In the event of a failure, the standby application has already retrieved the information from the remote session and is ready to assume the active role immediately. In the case of the standby application backing up numerous active applications, such as would be seen in an N+M redundancy model, the standby application may choose to only consume data from a checkpoint session when the standby has received an update to assume the active HA state.

The Element Checkpoint Service supports hierarchical, table-based organization of data records. This simplifies management of the checkpoint data store as well as enables prioritized retrieval of data from the checkpoint session.

**The Element Software Management Service**

The Element Software Management Service provides a full array of image management and upgrade features, both for cluster-wide restart upgrades (or “cold” upgrades) and for in-service upgrades (or “hot” upgrades). The Element Software Management Service is integrated with the Element AMF for the purpose of performing in-service upgrades.

The image management capabilities include the ability to discover and verify installed software releases, as well as replicate the validated software releases to a redundant software management server. Tools are included to generate pack-
The upgrade campaign in an in-service upgrade is governed by configured upgrade sequence rules. These rules indicate the restart order for each entity (i.e. nodes, service providers, or applications) and whether entities can be restarted in parallel or in serial. Each serial step in the upgrade sequence is gated by the cluster readiness, determined by a cluster-wide simple readiness service that aggregates individual application readiness. This allows the upgrade steps to progress no faster or no slower than necessary. There are configurable timeouts to prevent against infinite hangs in upgrade sequencing if the cluster doesn’t return to ready state. This is handled as an upgrade error condition.

All of the entities in the system are described in the upgrade sequence rules. This provides a simple, automatic, design driven generation of the upgrade campaign, which acts on the subset of entities that need to be restarted based on the delta from the current release to the target release. This simplifies the creation of the upgrade campaign, reducing the possibility of errors.

In the case of an error in the upgrade procedure, the upgrade manager will restart the cluster. If the cluster is restarted or rebooted prior to committing the target release, the cluster will restart on the original software release. This provides a dependable return to stability. After a commit to a new release, the cluster will stay on the target (now current) release until the next upgrade.

Conclusion

The Element family of middleware frameworks provides the necessary capabilities to deliver scalable, manageable solutions that are highly available. In addition these frameworks enable fast time to market of solutions that are maintainable.