z/TPF SOA White Paper

Document Version: 1.2
Date: April 28, 2006

Editors:

Barry Baker
IBM TPF Lab, Poughkeepsie, NY

Bill Cousins
IBM TPF Lab, Poughkeepsie, NY

Table of Contents

1 Introduction

TPF customers have made extensive investments in systems and application resources over the course of
many years and they have large amounts of business logic and application data stored on and managed by
TPF. Service-Oriented Architecture (SOA) refers to an architectural solution that creates an environment

where services, service consumers, and service providers can coexist, and still have no dependence on each other. SOA enables an enterprise to increase the loose coupling and the reuse of frequently used software assets. These software assets, together with the functionality that they provide, are called services in the SOA terminology.

For most enterprise-level IT organizations, the path to SOA will take time and be accomplished by incremental change that provides both short-term and long-term value. Major replacement projects are risky and expensive, and are warranted only when the existing systems no longer satisfy the business needs. Moving incrementally toward SOA will cause little disruption to the systems, and, when properly planned, requires a minimal investment in staff skills to make the changes possible.

In the enterprise, SOA can provide two types of value: access and reuse/flexibility. A new application being developed in this environment would be able to provide for both types of value, while an existing application undergoing SOA enablement can follow one of three patterns, wrapping, refacing, and componentizing, depicted in Figure 1.

Figure 1: SOA Enablement Patterns for Existing Applications

Figure 1 shows the three patterns for transforming an existing application, noting that as you move to the right, both the amount of work required and the payoff increase. Each IT organization will need to analyze their application assets and determine which pattern, with its associated costs and benefits, align with its business goals.

SOA allows for the reuse of existing assets where new services can be created from an existing IT infrastructure of systems. In other words, it enables businesses to leverage existing investments by allowing them to reuse existing applications, and promises interoperability between heterogeneous applications and technologies. Some key aspects to SOA that make it flexible include:

- Services are software components that are exposed through implementation-independent interfaces
- Services perform predetermined tasks and are loosely coupled
- Services can be combined into composite services
- Services can be dynamically discovered and then used.

With regard to the SOA programming model, three concepts will be introduced here to provide context: Service

Component Architecture (SCA), the Enterprise Service Bus (ESB), and Service Data Objects (SDOs).

SCA is a set of specifications that describe a model for building applications and systems using a service-oriented architecture. SCA extends and complements prior approaches to implementing services and provides an environment for components to operate in. SCA is not required to apply the componentizing SOA enablement pattern mentioned above, but it does help with the reassembly and exposure of created components. One of the mechanisms SCA utilizes for wiring service components together is Web services.

The following quote is a concise definition of the ESB, taken from the IBM Systems Journal, Vol. 44, No. 4, 2005, written by M.-T. Schmidt, et al:

The ESB enables an SOA by providing the connectivity layer between services. The definition of a service is wide; it is not restricted by a protocol, such as SOAP (Simple Object Access Protocol) or HTTP (Hypertext Transfer Protocol), which connects a service requestor to a service provider; nor does it require that the service be described by a specific standard such as WSDL (Web Services Description Language), though all of these standards are major contributors to the capabilities and progress of the ESB/SOA evolution. A service is a software component that is described by meta-data, which can be understood by a program. The metadata is published to enable reuse of the service by components that may be remote from it and that need no knowledge of the service implementation beyond its published meta-data. Of course, a well designed software program may use meta-data to define interfaces between components and may reuse components within the program. The distinguishing feature of a service is that the meta-data descriptions are published to enable reuse of the service in loosely coupled systems, frequently interconnected across networks.

Note that the ESB is the infrastructure for interconnecting services, but the term ESB does not include the business logic of the service providers themselves not the requestor applications, nor does it include the containers that host the services. Hosting containers and free-standing applications are enabled for interaction with ESBs with varying levels of integration, depending on the range of protocols and interoperability standards supported.

The ESB implementation for TPF will be defined more precisely (that is, restricted) and support the Web services form of SOA only. The details of the makeup of the TPF ESB are presented further in this paper, but basically the ESB will consist of the following support: the protocols that are planned to be supported will be restricted to SOAP/HTTP, SOAP/HTTPS, SOAP/MQ, and SOAP/MQ over SSL (protocols that may be considered for future support are SMTP, FTP, and IIOP); WSDL will be the standard to describe the metadata for the services; a TPF-specific design will be included for services

deployment; lastly, message handlers and drivers/stubs will be employed to provide mediation services for message manipulation (for example, data translation and encryption/decryption) and enable the ESB to be extendable for future support for additional Web services standards.

The container spoken of in the second quoted paragraph is the TPF application space. It is supported by new APIs used to access ESB services (described below) and the current TPF API set. These ESB APIs will be used by the service running in the TPF container

SDOs simplify data access and representation in your service-oriented software. SDOs replace diverse data access models with a uniform abstraction for creating, retrieving, updating, and deleting business data used by service implementations. They make developers more productive by freeing them from the technical details of how to access particular back-end data sources, so they can focus principally on business logic. SDOs define a single and uniform way to access and manipulate data from heterogeneous data sources including relational databases, eXtensible Markup Language (XML) data sources, Web services, and enterprise information systems (EIS). The TPF statement of direction is to expose TPFDF databases to application environments off of TPF (for example, WebSphere Application Server) using SDOs

SOA and Web services are two different things; SOA is an architecture, while Web services is one way to implement the SOA architecture. Web services is becoming the preferred way to realize SOA due to its extensive use of open standards. Web services are software assets designed to support interoperable machine-to-machine interaction over a network. This interoperability is gained through a set of XML-based open standards, such as WSDL, SOAP, and UDDI. These standards provide a common approach for defining, publishing, and using Web services.

There is very clear evidence that Service Oriented Architecture (SOA) based on Web services represents a shift in the dominant enterprise software development and deployment paradigm, as well as the evolution of enterprise application integration (EIS) solutions (for example, WebSphere MQ). Although this shift is at an early stage, there is sufficient evidence that it will have a major industry impact over the next five years. This evidence clearly shows that Web services is rapidly shifting from emerging technology to the mainstream.

1.1 Background

This paper is an evolutionary progression of the AAA Task Force Objectives paper dated March 7, 2003. That paper introduced Web services concepts to the AAA Task Force, discussed conditions that led to the formation of Web services, and proposed how Web services could be implemented on TPF. It

was hoped that the Objectives paper would engender greater interest in developing Web services on TPF and create requirements for the TPF product so that IBM could deliver enhancements to aid in the Web enablement of existing applications. Indeed, several customers implemented their own in-house solutions to accomplish this goal (for example, wrote their own HTTP server, SOAP server, XML parser, etc.) Unfortunately, no requirements were forthcoming, probably due mostly to the fluid (at the time) environment of Web services, the confusing number of “standards” being published, and some lack of understanding of the overall concepts. The path to SOA and Web services is by now, a well-trodden and well-defined one, and this is the time for IBM to put into motion a plan to provide an enterprise-class SOA/Web services solution.

1.2 Purpose

The purpose of this document is to show how z/TPF can participate in a unified IBM/SWG strategy, thus allowing it to be a part of SOA solutions through the use of Web services. The scope of this document includes sections that introduce the concepts and components necessary to implement this support on z/TPF. There are three primary components necessary to support an SOA environment: Provider services which allow z/TPF to be a host site for services; Consumer services which allow applications running on z/TPF to access Web services on remote hosts; and Tooling services which assist the z/TPF application developer to more easily architect solutions for the SOA environment. This support will support the three SOA enablement patterns mentioned above, but will not provide explicit information about how to deconstruct an existing application into components and then reassemble them as is required in the componentizing pattern. This is left to the customer (and potentially a future paper to provide information on available tooling), but once this work is done, the processing and tooling described here can be used to expose the newly created components as Web services.

Also documented in this paper are the internal specifications and APIs that will form the basis of SOA support in z/TPF. Part of this design is an SOA Engine that is intended to implement on z/TPF a framework that is common with other SWG products (such as CICS, DB2, and IMS) that will allow code sharing among these products when new requirements are implemented. These APIs assist in standardizing satellite components such as the Message Handler, Parser, Service Registry, Transport Handlers, and Service Wrapper/Stub/Application. The primary components and satellite components complete the SOA for z/TPF implementation picture.

1.3 Current SOA Support on z/TPF

Processing a SOAP message on z/TPF involves a variety of components. The flow of a SOAP message through the z/TPF system varies depending on the SOAP message itself, whether each

component is able to complete its
portion of the processing, and any special coding in the appropriate
user exits.
The following figure summarizes the components through which a SOAP
message flows and the paths that it
may or may not take:

SOAP communications binding
A communications binding on z/TPF receives a SOAP message from the
network. The binding calls the
tpf_soap_handler C function to pass the message to the SOAP handler.
When SOAP message processing is
completed, the communications binding receives the response message from
the SOAP handler and sends
the message to the SOAP client.
Currently, the Apache HTTP server is available for the customer to
install on their z/TPF system. Apache
is able to process SOAP messages only after the mod_tpf_soap piece of
the Apache program has been installed,
which enables recognition of a SOAP message and passes it to the SOAP
handler. The Apache HTTP server and the
mod_tpf_soap program for Apache are not included in the z/TPF base.
Note:
SOAP support on the z/TPF system is not compatible with the version of
the Apache HTTP server with SSL support.

SOAP handler

Network
SOAP
Communications
Binding
SOAP
Handler
tpf_soap_appl_handler
user exit
TPF
Application
B2B
XML
Scanner
INode
Table
tpf_soap_handler_exit
user exit
tpf_soap_build_fault
C function
Translation
Functions
XML APIs
The SOAP handler is a program that receives a SOAP message from HTTP or
other communications binding on the
z/TPF system and performs some or all of the following tasks, each
followed by a return before moving on to
the next appropriate task:
• Calls the SOAP handler user exit (tpf_soap_handler_exit) for any user-
specific processing

• Calls translation functions, if needed, to ensure that the SOAP message is in the host encoding

• Passes the SOAP message to an XML scanner

• Performs the SOAP syntax checks.

• Calls the tpf_soap_build_fault C function, when needed, to create a fault message or passes information to the SOAP application handler user exit (tpf_soap_ appl_handler)

• Returns the appropriate response to the HTTP server or other communications binding to send to the SOAP client.

SOAP handler user exit

The tpf_soap_handler_exit user exit is called by the SOAP handler. This user exit, by default, simply returns the SOAP message unchanged. Customers can use this user exit for any user-specific instructions that are appropriate for their system. For example, they may need to add some translation routines in this user exit to ensure that the SOAP message is in the proper host encoding

B2B XML scanner

The B2B XML scanner receives the SOAP message from the SOAP handler. The scanner is an internal component that processes a SOAP message to create a series of structures that are in the cbnode.h header file. This header file is essentially a copy of the SOAP message in a structured, tree format. Once this processing is completed, the B2B XML scanner returns to the SOAP Handler.

Note:

Application programmers can use z/TPF XML API support to access data from the request and to create response messages without having to interact directly with the structures in the cbnode.h header file.

The SOAP message is changed by the B2B XML scanner to normalize the message. Normalization is a normal parsing process and is defined by the W3C on their Web site at http://www.w3.org/. The changed SOAP message cannot be parsed again after the message has been normalized by the XML scanner.

Changes made to the SOAP message conform with the W3C specification with the following exception:

If a character reference cannot be represented in the encoding of the SOAP input message encoding, it is handled as text and remains unchanged.

The XML scanner does not validate the XML structure.

SOAP application handler user exit

The tpf_soap_appl_handler user exit is called by the SOAP handler. This user exit is used to specify how the data contained in the SOAP message is processed, including passing that data to an application on the customer’s z/TPF system.

The customer can call the following components as needed.

Fault builder

The tpf_soap_build_fault C function builds an XML-based fault message when an error occurs along the SOAP message path. This fault message is returned to the SOAP client.

A fault message is built for each of the following conditions:
• An error occurs during SOAP application processing. (The SOAP fault is built with identification of either
receiver or sender error.)

• A SOAP translation error during SOAP handler processing.

• The SOAP message sent by the client arrives at z/TPF encoded in a character set other than UTF-8, Latin 1
(ISO-8859-1), or EBCDIC and tpf_soap_handler_exit does not translate the message.

• When tpf_soap_handler_exit detects an error in the input message, a SendErrorReplySender return value must
be set to return to caller.

• When an application handler returns with ErrorReplyNeeded, the SOAP handler will build a fault message with
faultCode Receiver to return to the client.

The tpf_soap_build_fault C function can also be called to build a fault message by other components or
applications at any time.
z/TPF application
This is an application that customers will supply on their z/TPF system to handle SOAP messages. It may or
may not call translation functions, the tpf_soap_build_fault C function, or any other supported z/TPF functions.
When translating from Unicode to the single-byte EBCDIC character set, a substitute character replaces some
Unicode sequences that are not valid (sometimes referred to as illogical sequences). A customer’s application
must be able to take correct action on messages that have been translated from UTF-8 to EBCDIC. The correct
action is application- and data-specific.
Translation functions
Translation functions allow the customer to translate a message from one character set to another. Generally,
a SOAP message must be in the host encoding before being processed by a z/TPF application. If a SOAP message
is sent to the client in response, it also must be encoded in the preferred character set of the client
before being sent.
When the input message is encoded in a Unicode character set but contains characters that cannot be translated,
a substitute character is used. For example, a euro symbol (U+20AC, which is represented as 0xE282AC in UTF-8)
in an input message encoded in Unicode format UTF-8 will be replaced by the 0x3F EBCDIC substitute character
during translation. The converted message will instead have the 0x3F EBCDIC substitute character

1.4 SOA Myths: [SOA Web Services Journal]

With the introduction of any new architecture or technology, myths about its abilities, costs, and placement
in the current IT spectrum are created, spread, and end up setting

expectations. Years later, when the expectations created by these myths go unmet, the relationship between the business and IT organization are damaged. This can cause the IT organization to be isolated and marginalized by the business. The following section attempts to skewer some of the common myths that are unfortunately taking hold in the community, and to ensure a greater linkage between the business and IT organizations.

Myth #1: SOA is a solution (panacea) to all software problems
SOA is an architectural approach used to build solutions that are characterized by the presence of a set of services, service consumers, service providers, and service contracts. The approach of SOA needs to be used in light of business processes to arrive at a solution that can provide business benefits. Though SOA provides a sound architectural foundation to the overall solution, the specific problem regarding domain/business needs to be solved using the domain/business expertise, over and above the SOA solution.

Myth #2: SOA is like a product, and can be downloaded for trial
SOA is an architectural approach for building solutions that are loosely coupled in a stepwise, phased manner, resulting ultimately in the realization of a complex, federated, service-oriented enterprise. The business-specific services are initially identified over the SOA architecture, and then mapped to a set of technology-specific implementation architectures for the purpose of realization.

Though SOA concepts are reasonably simple to understand and apply, it is a rather involved process to build an SOA-rich enterprise, covering all aspects of the SOA characteristics. These solutions evolve over time, and they need to be crafted carefully and jointly along with the customer to ensure that the journey in the SOA architecture evolution is progressing in the "correct" direction. Due to this very nature, many SOA solutions typically do not fall into the category of "products."

Myth #3: SOA is a complete, off-the-shelf solution
SOA solutions are composed of prefabricated building blocks that typically represent the services identified during architecture workshops. The concept of prefabricated building blocks (services) reduces time to market, risk, promotes reusability, and provides a head start. While the generic, technology-neutral SOA approach continues to strive for increasing levels of reusability, the technology-specific SOA solution always requires some degree of customization. The degree of customization is based on factors including if the customer's environment already has a set of services or an environment with different degrees of legacy applications and integrations. The service-oriented approach provides a significant differentiation in building a federated service-oriented enterprise, and helps in realizing the business services reasonably quickly.

Myth #4: SOA software always needs to be developed using Web services
SOA is a technology-neutral architecture, and can be realized using any technology. The selection of
technology is performed by considering the various possible factors such as the functional requirements
to be addressed, the performance and reliability requirements, the available budget, and so on. Based on
these factors, the technology is chosen. Web services offer just one such technology option that is used
to realize SOA solutions. However, it is possible to use other alternatives (apart from Web services), and
still realize SOA solutions.

Myth #5: Any software development using Web services is aligned with SOA
Web services, coupled with the other relevant tools and technologies, offer one option that can be used to
build and realize an SOA solution. However, a solution cannot be classified as SOA just by virtue of being
built using Web services. A solution is compliant with SOA if it meets the following requirements:

- Interaction between service providers and consumers
- Usage of service contracts
- Usage of metadata.

This can be compared to object-oriented architecture (OO). It is possible to use an OO language such as C++
and still end up in non-OO architecture, unless the necessary characteristics of an OO solution are addressed
while building the solution.

Myth #6: Each service is always atomic in nature
Services in the context of SOA represent the functionality provided by software assets. These services,
when invoked, perform a specific task. At the lowest level, these services are mapped to a specific task.
The services that always perform one atomic task are referred to as "leaf" services. The services that
are created by the federation of other services are called "composite" services. In other words, it is
possible to define services in the SOA context that are, in turn, composed of other SOA atomic services.

Myth #7: SOA is not aligned with any standards
SOA is based on several industry-standard initiatives, namely the OASIS working group, the Web services
standards bodies, and so on.

Myth #8: SOA is the same as EAI There is a general misconception that SOA is the same as enterprise
application integration (EAI). EAI is the integration approach in which various applications are integrated
using a middleware, through the use of a set of connectors (or adapters). These adapters provide access to
and exposure of all of the atomic interfaces of the underlying applications.

However, SOA is not the same as EAI. SOA is based on service aggregation that is based on functionality,
and not on atomic APIs. SOA can be visualized as a further evolution of EAI.
SOA advocates integration based on services rather than on atomic APIs. SOA integration is similar to a
richer form of ESB (enterprise service bus) integration, and represents a significant evolution from traditional
EAI integration. Using SOA as an architectural approach results in significant improvement in the performance,
flexibility, usability, and TCO (total cost of ownership) of the overall solution.
SOA is more sophisticated than "classical/traditional" EAI in several ways. First, SOA provides an aggregation
capability (support for composite services) that is lacking in EAI. EAI deals with basic atomic APIs and data.
Second, SOA provides support to work with service-level data, whereas EAI always deals with application
integration using atomic API (application programming interface). Also, most important, SOA provides support
for transformations and mappings, whereas EAI does not support these directly. Keeping all this in mind, it
is possible to say that SOA is a more advanced architectural methodology.
Myth #9: SOA is a very expensive solution
SOA solutions are deployed in an evolutionary, stepwise manner that requires incremental investments.
However, the framework allows for the consistency across the incremental solution.
The cost of the solution depends on several factors, among which the level of automation and the level
of sophistication required in the solution are foremost. It is possible to arrive at a reasonable level
of automation, and design and build an SOA solution that is cost-effective. Also, the cost depends on the
choice of the other parameters such as the technology chosen, the products chosen (in case of green-field
customers), and so on. All of the factors that contribute to the cost need to be considered carefully,
and appropriate choices need to be made in order to reduce the cost. By doing so, it is possible to build
a reasonably feature-rich and yet cheap solution. The enterprise architecture plays a crucial role in the
SOA roadmap for the enterprise and precedes any major commitments. The concept of service and a means of
interaction are more important than changing technologies overnight.
Myth #10: SOA solution components (services, contracts, and data model) are completely reusable
SOA strives for the highest possible amount of reuse, and the amount of reuse achievable increases over
time.
In terms of the service, a large amount of reuse is possible in the technology-neutral representation.
However, as the implementation is associated with the chosen technology, the reuse is limited if the
technology is changed. However, when newer services are designed using existing services, a large amount
of reuse is possible. In any case, the learning and the knowledge can definitely be reused, in addition
to possible code reuse.

2 z/TPF Service Provider Support

The service provider creates a Web service and possibly publishes its interface and access information to the service registry. Each provider must decide which services to expose, how to make trade-offs between security and easy availability, how to price the services, or, if they are for free, how to exploit them for other value. The provider also has to decide what category the service should be listed in for a given broker service and what sort of trading partner agreements are required to use the service.

Customers using z/TPF can leverage their existing legacy applications by using the z/TPF SOA/SDO support to transform them into Web services, accessible by clients and hosts using an Intranet/Internet connection. Indeed, customers are now able to include the z/TPF system in their application solutions architected for a Web services environment.

2.1 z/TPF Web Services: Server/Provider

Figure 2: TPF Web services Support - Server/Provider Side: Structure Diagram

This figure attempts to show a structural diagram of the server/provider side of Web services support that is either already available in z/TPF (shown in green) and the proposed additions (shown in red). As z/TPF implements the proposed additions, the current path through the support will be maintained to not affect current users. Currently, the tpf_soap_handler passes requests up to the user-implemented tpf_soap_appl_handler. The main function of the tpf_soap_appl_handler is to route requests to the appropriate application, and the applications would then be responsible for accessing the request information from the output of the B2BScanner (infonode structure instance), invoke the application, and build the SOAP response message. The proposed support would rely on the deployment mechanism (which relies on the Web service deployment descriptors and indirectly the Web service description) to, among other things, perform the necessary SOAP Header processing (via the message handlers), and the routing of requests to a particular Web service driver that is used for exposing one Web service. The Web service driver would be responsible for, and thus shield the application from having to be updated to, accessing the request information, invoke the application, and build the Body of the SOAP response message. After this Body is built, the message handlers would create the necessary SOAP headers. This architecture

provides for the isolation of where
certain new skills may be required. For example, the functionality that is contained in the various Web
service drivers requires skills and knowledge in areas of SOAP and XML, which may be skills that current
application owners do not have. By isolating where these new skills are required, you limit the number of
developers that you may need to train in those new skills.
3 z/TPF Service Consumer Support

The service consumer (or Web service client) locates entries in the broker registry using various find
operations and then binds to the service provider in order to invoke one of its Web services. A client
is not coupled to a server, but to a service. Therefore, the integration of the server takes place outside
the scope of the client application programs.
3.1 z/TPF Web Services: Client/Consumer

Figure 3: TPF Web services Support - Client/Consumer Side: Structure Diagram

This figure attempts to show a structural diagram of the client/consumer side of Web services support that
is proposed for z/TPF (shown in red). The proposed support would provide applications with a SOAP Client
(APIs) guided by input and feedback from the TPF Users Group since the only standard API in this area is
in Java ™ (for example, JAXM, JAXR, and JAX-RPC). The application that wishes to consume a Web service
would instantiate a Web service handle (using the TPF SOAP Client) and requests to call a particular service
by name. The TPF SOAP Client would then inspect the deployment mechanism to determine if this z/TPF
system has been configured to provide access to the requested service. If the z/TPF system has been
configured to support this service, the corresponding message handlers and Web service stub name would
be provided to the TPF SOAP Client. The application would then be able to initiate the consumption of
the requested service. To satisfy the request of the application, the TPF SOAP Client would rely on the
Web service Stub to take in the parameters from the application and build the Body of the SOAP request
message. After the SOAP Body has been constructed, the TPF SOAP Client would use the message handler
list (returned from the deployment mechanism) to direct the building of the SOAP Headers and the complete
SOAP request message. The SOAP request message would then flow through a transport to the remote service
provider. The response from this request would perform the reverse of this flow through the message handlers
to correctly process the SOAP Header and then through the Web service stub to perform the data transformation

from XML format to the structures used by the application.

4 Artifacts

A discussion of artifacts should begin with the higher level discussion of a programming model. How IBM's Software Group defines programming model in the context of a Service Oriented Architecture (SOA) is summarized below:

A programming model is central to IBM SOA and IBM products in general. It defines the concepts and abstractions that developers build and use. Runtime products run or host the programming model artifacts. Development tools support the modeling and implementation of programming model artifacts, their assembly into applications (solutions), and their deployment into the runtimes. Finally, systems management products, agents, and instrumentation support the administration of the runtimes and the programming model artifacts they host.

What is a programming model? Although there is no generally accepted definition, we like to define it as:

* A set of part types that programmers build. Part types encompass the diversity of programming model artifacts.

* A set of roles that groups members of the development and administrative community who have similar skills and knowledge. Categorizing developers in this way helps produce role-appropriate tools that enable non-programmers to implement services and assemble solutions from services. Each role contains:
    - o Skills that the role possesses.
    - o Part types and application interfaces with which the role interacts.
    - o Tools that the role uses.

The typical z/TPF application consists of a collection of shared objects (SOs) that may call/utilize user and/or system libraries, or call/use system services. After allocating and loading these new SOs to a z/TPF system there may be additional work or activities to make the application accessible to users. For example, if the application receives input via TCP/IP connectivity, then you may need to define a new server to the Internet daemon; for terminal-based applications, you may need to update system tables (for example, ANT, RCAT) used by COMM SOURCE to route input messages to the appropriate application, or for applications that rely on z/TPF MQ Series middleware, you may need to define a local queue to the z/TPF system (if the queue is to reside on z/TPF). Beyond making the application accessible on z/TPF, users of the application will need to be made aware of the specifics of where the application

is and how to access the application (that is, message format). With regard to artifacts, the current z/TPF application (the source code and the built SOs) can be considered artifacts as well as the various server definitions, system tables, and local queue definitions.

As customers move to a service-oriented architecture (SOA) through the use of Web services, among other technologies and techniques (for example, Enterprise Service Bus, service component architecture (SCA), service data object (SDO)…), new concepts, activities, and artifacts will be defined and performed. At its core, Web services technologies provide a common abstraction layer focused on dealing with interoperability between services. To achieve this, Web services specifications provide a standard way of constructing and integrating applications using XML-based open standards over an Internet/intranet backbone.

As z/TPF evolves to participate in an SOA, there are some issues that should be kept in the forefront. The primary issue is that Web services technologies have been developed in the context of Java™ and to a lesser extent, or inherently, object-oriented programming. These two contextual points do not preclude z/TPF from implementing Web services specifications and playing a role in an SOA, but they do affect the roadmap, and some of the specifics, of how z/TPF will implement the various specifications. Because of these issues, z/TPF's adoption of the various Web services specifications may be very specific to the particulars of z/TPF and may not be able to take advantage of or use commonly available solutions/tooling since the majority of these solutions/tooling assumes a Java™ based runtime. Considering this context, z/TPF will incorporate the Web services technologies in such a way as to NOT introduce new concepts, activities, and artifacts beyond those explained in the commonly accepted SOA programming model. The intention is to not require developers working on the z/TPF platform to have to obtain any special skills that are not required for the developers working on other platforms that are participating in an SOA.

4.1 z/TPF Web Services: Server/Provider

Figure 4: TPF Web services Support - Server/Provider Side: Structure Diagram

This figure, which was already introduced in Section 2, attempts to show a structural diagram of the server/provider side of Web services support that is either already available in z/TPF (shown in green) and the proposed additions (shown in red). The additions in red consist of new components (the z/TPF XML

API and the deployment mechanism) and artifacts. The following is the list of artifacts that may need
to be created and deployed by someone who wants to expose an application as a Web service:

- The application (source code and SOs)
- Web Service Driver
- Message handler(s)
- Web Service Deployment Descriptor
- Web Service Description (WSDL)

4.2 z/TPF Web Services: Client/Consumer

Figure 5: TPF Web services Support - Client/Consumer Side: Structure Diagram

This figure, which was already introduced in Section 3, attempts to show a structural diagram of the
client/consumer side of Web services support that is proposed for z/TPF (shown in red). The additions
in red consist of new components (the TPF SOAP Client and the deployment mechanism) and artifacts. The
following is the list of artifacts that may need to be created and deployed by someone who wants to
consume a Web service:

- The application (source code and SOs)
- Web Service Stub
- Message handler(s)
- Web Service Deployment Descriptor
- Web Service Description (WSDL)

4.2.1 SOAP Message Flow

From the previous structural diagrams, it can be hard to understand the flow of a SOAP message through
the z/TPF system. The following figure attempts to address this:

Figure 6: SOAP Message Flow

The figure covers both consumer and provider perspectives of SOAP support. On one end of the diagram

there is a SOAP message and on the other end is an application. The SOAP support can most simply be thought of as processing the transforms SOAP messages into native data structures (and back), and also performs routing of SOAP requests to and from the z/TPF system. The z/TPF SOAP Handler and the collection of message handlers that are used for a particular Web service are responsible for SOAP message validation and handling any of the extensibility points defined in the Header portion of the SOAP message. Note that this figure shows the message handlers as comprising a “pipeline” between the z/TPF SOAP Handler and either the particular Web service Driver or Stub, but the architecture could easily be one in which the z/TPF SOAP Handler directs the calling of each of the Message Handlers for a particular Web service. Once through the z/TPF SOAP Handler and the set of message handlers, the SOAP message is passed along to the Web service driver that is defined for a particular Web service, which is responsible for building a data structure or parameter data that can be used to activate an application.
The response on the provider side flows back through the Web service driver building the Body portion of the SOAP response message, followed by the message handlers and the z/TPF SOAP Handler, which are responsible for building the necessary Header portions of the SOAP response message and building the complete SOAP message and delivering it to the appropriate transport. The consumer side is just the opposite flow, starting with the Application building a data structure that is known to a Web service stub, which is responsible for building the Body portion of the SOAP request.

4.2.2 Static versus Dynamic Client/Consumer

The type of Web services Client described previously is considered static because, before an application can consume a particular Web service, offline tasks must be performed to create both a Web service Deployment Descriptor and a Web service Stub. The other end of the Web services Client spectrum would be considered dynamic because such a Client would allow the application to either search for a particular service based on some required functionality metadata and be able to use a Web Service Description (WSDL) to direct the formatting of the SOAP request message and correctly handle the SOAP response message. The two main reasons for using a dynamic approach are to 1) be able to handle frequent changes to a particular Web Service Description (WSDL) and 2) to be able to select a service provider at runtime.
At the time of this writing it is not believed that applications for z/TPF would make use of a dynamic client due to the response time constraints of the typical z/TPF application, but this is not to say that it would not be considered in the future based on customer needs.

Sections 5-8 will provide further detail about each of the previously

mentioned artifacts.
5 Message Handlers

As described previously, the message handlers are responsible for processing the different elements
found in the Header portion of a SOAP message (see figure following).

```
<soap:Envelope xmlns:soap='http://www.w3.org/2001/10/soap-envelope'>
    <soap:Header>
                            <-- Headers go here -->
    </soap:Header>
    <soap:Body>
                            <-- Request/Response goes here -->
    </soap:Body>
</soap:Envelope>
```

SOAP Message Format: Header

The Header element in SOAP provides an extensibility mechanism. This element can contain any number of
namespace qualified child elements. Each of these elements is some form of extension to the base SOAP
protocol. Perhaps one element contains data associated with conversation or session management between
a client and a server. Another element might contain authentication information or even information
pertaining to an ongoing transaction. Whatever their content or semantics, each header element modifies
the SOAP protocol in some way, providing extra context for the processing of the body of the message.

Beyond the standard extensions to the SOAP protocol, the user of z/TPF SOAP support can develop and
deploy custom message handlers to perform installation/environment specific processing. For example,
if a set of deployed Web services requires that each Request for them must be logged to a certain logging
facility, this can be done by developing a logging message handler and deploying it to z/TPF SOAP support.

A message handler will consist of:
1. C source that implements the processing to be performed per message

2. A deployment descriptor that is used to deploy the message handler to z/TPF SOAP support and make it
available for use by deployed Consumer/Provider Web services.

The C source that implements the message handler will be bound by a specific interface and will look
similar to the interface for the TPF SOAP Handler:

The message handlers will operate on the soapMsg structure passed in on

the inputMsg parameter or the
outputMsg parameter based on the direction of the flow through the handler, and either the commsBinding
structure will be enhanced or a new structure will be created to help control the processing of the
message handlers. What is needed is a way to allow for a message handler to build and return a SOAP
fault message, if necessary, and signify to the other message handlers that may be encountered that a
SOAP fault has occurred so that those message handlers can react appropriately.

5.1 Message Handler Tooling

The tooling that would assist the development and deployment of message handlers consists of a template
generator to create the structure of the C source that implements the message handler and a deployment
descriptor generator used to deploy the message handler to z/TPF SOAP support.

In the TPF Toolkit, a new project type of Web service Message Handler could be created with a corresponding
wizard that takes, as input, a message handler name, a 4-character program name, and a high-level
description of the Web service Message Handler and generates the C source template and the complete
deployment descriptor.

When a Web service message handler has been coded, the TPF Toolkit could also provide a mechanism to
deploy the message handler by helping with the loading of the message handler module and FTPing the
deployment descriptor to the z/TPF system and issuing a Z-command to update the deployment mechanism.

It is important to note that the number of developed and deployed message handlers is expected to be on
the order of 10s, thus enabling this portion of tooling to be easily replaced with minor documentation
and sample templates. See the Tooling section of this document for further information.

6 Web Service Driver

As described previously, at a high level the Web service driver is responsible for transforming the
Body portion of a SOAP (see figure below) request message into a representation that is more readily
usable by the z/TPF application. When it has done this, the z/TPF application can be invoked. Upon
return, the Web service driver is responsible for starting the building of the SOAP response message,
specifically transforming the z/TPF application response data into the Body of the SOAP message.

There is a one-to-one mapping between deployed Web services and Web service drivers.

```
<soap:Envelope xmlns:soap='http://www.w3.org/2001/10/soap-envelope'>
    <soap:Header>
                            <-- Headers go here -->
    </soap:Header>
    <soap:Body>
                            <-- Request/Response goes here -->
    </soap:Body>
</soap:Envelope>
```

SOAP Message Format: Body

The Body portion of the SOAP message contains the information that is specific to a particular operation of a Web service. A Web service can be a collection of many operations and their respective messages that define the structure of the input and output data. A Web service name is the high-level name (for example, name on the URI for HTTP) that is used by the z/TPF SOAP Handler to query the deployment mechanism to obtain the list of message handlers that a SOAP message should flow through and its associated Web service driver.

The Web service driver will be responsible for accessing and converting all of the required data in the Body portion of the SOAP message into a native format that can be used by the application, and it will have to inspect the Body data to determine the requested operation and call the correct application. Here we start to get into the dirty details of the various formats that are possible for the Body. The various formats that are possible for the Body are known as WSDL Styles. At the time of this writing, the WS-I is working to standardize on one of the WSDL styles to ease the creation of interoperable SOAP runtimes and Tooling support. The current style that is expected to become the standard is Document/Literal Wrapped (Doc/Literal wrapped is the interoperability leader supported by Apache Axis, gSoap, Websphere, and .NET). The following shows an example Body that uses Document/Literal Wrapped is shown below:

```
<soap:Body>
        <myOperation>
            <x>5</x>
        </myOperation>
</soap:Body>
```

Sample Document/Literal Wrapped SOAP Body

As you can see from this example, the SOAP Body will contain an element that is named for the operation that is requested (myOperation in this example). This element will contain children elements that define the parameter date that is to be passed to the application that corresponds to myOperation. The Web

service driver for this Web service will consist of processing that accesses the operation name from the SOAP Body, and then accesses the parameter data that is required based on the operation name and translates it into a C datatype that is usable by the application. After this is done the application is called, and when it returns to the Web service driver, the driver is responsible for creating the Body of the SOAP response message based on the definition of the response message as described in the WSDL for the operation, and then returns down the chain of message handlers as discussed in the previous section.

As you can see, the Web service driver is dependent on the WSDL for the Web service and the interface and data structures of the application. At the time of this writing, WSDL 2.0 is considered a Candidate Recommendation, slowly approaching the final Recommendation status, and the timeline of this development will need to be considered when implementing support that is strictly based on WSDL.

The “bottom-up” approach to deploying a Provider Web service consists of taking an existing application and making it available as a Web service where a WSDL for the Web service does not exist. The ultimate goal is to require no application changes to deploy an application that already exists. To achieve this, the user would want tooling to be able to inspect the interface of the application (that is, C parameters/structures) and generate the XML schema definition of the input and output messages for each of the operations. And you would want the Web service drivers to be generated to make calls to XML conversion routines that transform between the XML data in the Body of the SOAP message and the C parameters/structures. Because the XML Schema specification defines many different datatypes, some of which do not map to C/C++ datatypes, this tooling and runtime support will support a subset of the XML Schema datatypes.

The “top-down” approach to deploying a Provider Web service consists of starting with a pre-existing WSDL description of a service (of interest are the abstract portions of the WSDL, namely PortTypes, Messages, and Types for WSDL 1.1) and creating an implementation of the various operations defined to the Web service. For this case, the developer requires that there be tooling to help with the creation of skeleton code for the application and the interface information (C header file), including passed parameters/structures. The same restriction applies to the “bottom-up” approach that the tooling that helps to create the interface information from the abstract WSDL information will not be able to support all of the datatypes defined by the XML Schema specification.

6.1 Web Service Driver Tooling

For creating Web service drivers, tooling is essential if we are to succeed in limiting how much about the various Web services specifications we require customers to know. It is fair to require that customers understand the concepts contained in the specifications, but our goal should be to shield them from the details as is done on other SOAP platforms. Due to some of the z/TPF unique application interfaces, it is not expected that the tooling will be able to generate complete Web service drivers, but rather templates that are about 70-80% complete. See the Tooling section of this document for further information.

7 Web Service Stub

When the Web service driver has been defined, conceptually it is easy to understand what the Web service stub is responsible for. The Web service stub is used when z/TPF is acting as the Consumer of a particular Web service. The Web service stub performs the reverse processing of the Web service driver. The Web service stub will first be invoked by the application through the use of z/TPF SOAP support and it will build the Body portion of the SOAP request and then pass the request along to the chain of message handlers that are associated with the deployed Web service (Consumer). Upon return from the Provider of the Web service, the response message will eventually make its way through z/TPF SOAP support and message handlers, and will end up back at the Web service stub. The stub would then transform the Body portion of the SOAP response message into a data structure to be returned to the calling application.

There is currently no standard C/C++ SOAP API, so we will be introducing a z/TPF-specific one that will be modeled after those found in other languages. In general, it is expected that a SOAP Client/Consumer API would consist of a couple of APIs that are listed below (in the order that they would be expected to be used):

1. tpf_init_SOAPReq(): This API would be the first that a Consumer would use to initialize a SOAP request structure. One parameter would be a Web service name. This API would query the same deployment mechanism used by the TPF SOAP Handler on the Provider side. From the deployment definition returned by the deployment mechanism, the SOAP request structure would be populated with information about the deployed Web service, including the Web service stub name and the set of message handlers.

2. tpf_getopts_SOAPReq(): This API will allow the user to inspect the

returned SOAP request structure
to determine if there are any optional message handlers that can be used for this message. For example,
the Web service could be deployed as allowing for a logging message handler to be used optionally.

3. tpf_setopts_SOAPReq(): This API will allow the user to specify the operation that is being requested,
attach an area of storage containing the parameter data for the response, and to turn on and off the
optional message handlers associated with this deployed Web service.

4. tpf_send_SOAPReq(): This API results in the SOAP request to be completely built and sent to the
Provider. z/TPF SOAP Support will invoke the Web service stub that corresponds with the requested Web
service.

5. tpf_terminate_SOAPReq(): This API will be responsible for releasing any storage that was obtained
by SOAP Support to satisfy the request.

NOTE: Further investigation is needed to validate the completeness and conceptual underpinnings of
this API.

When the tpf_send_SOAPReq() API returns successfully, the SOAP request structure will point to an
area of storage containing the data returned to by the Provider.

The discussion about the relevance of WSDL to the creation of the Web service drivers also pertains
to the creation of the Web service stubs and will not be repeated here (see section 1.1.2 for more
information).

7.1 Web Service Stub Tooling

The Web service stub tooling is responsible for generating (completely) a Web service stub and a
C header file for a Consumer Web service deployed to z/TPF, similar to the “top-down” approach
described in the Web service Driver section. The Web service stub will contain code to perform
the reverse of the data translation that is performed for the Web service driver: upon calling
the tpf_send_SOAPReq() API, the Web service stub is activated to translate the data structure
defined in the generated header file for the Web service into the appropriate Body portion of
the SOAP request, and upon being activated with the SOAP response message, the Web service stub
will translate the response information in to a response data structure.

8 Web Service Deployment Descriptor

The Web service deployment descriptor is the artifact that ties all of the previously mentioned
artifacts together to define a completely deployed Web service (Consumer and Provider). There
will be a one-to-one mapping between a deployed Web service and a Web service deployment descriptor.

Figure 7: Web Service Deployment Descriptor

This figure shows what a Web service deployment descriptor (file name extension .wsdd) for a given
Web service contains. It maps a Web service name, MyService, to its supported transports, message
handlers, and the Web service driver or stub (depending on whether this deployment is a Consumer
or Provider deployment).

The deployment descriptors will be XML files that conform to an XML Schema format defined by z/TPF.
To deploy a Web service to the z/TPF SOAP support, all of the required code (i.e. message handlers,
Web service driver/stub, and application) will need to be loaded to the z/TPF system. Following
this, the deployment descriptor file for a given Web service must be loaded to a specific directory
on the z/TPF filesystem. Once in the z/TPF filesystem, a z/TPF command can be issued to load the
deployment descriptor in to the Deployment Mechanism. There will also be other z/TPF commands to
do things like deactivate a Web service, deactivate/activate all Web services, and display Web services.

8.1 Web Service Deployment Descriptor Tooling

The creation of Web service deployment descriptors should occur at the same time that either the
Web service driver or stub is being created for a particular Web service. The complexity of the
Web service deployment descriptors it not expected to be high, and much of the information needed
to create them will have already been provided for the creation of the Web service driver or stub.
The XML schema definition of the deployment descriptors will need to be loaded to both the TPF
Toolkit (probably just included in the installed version of the TPF Toolkit because its format is
entirely controlled by IBM) and to the z/TPF filesystem to be used for validation of created and
loaded deployment descriptors. Regardless of the complexity of the deployment descriptors, it
would not be a good idea to push the creation of them entirely on to the user because the format
of them is specific to z/TPF. Other Web service runtimes utilized deployment descriptors and, in
the J2EE space, their format is standardized; but because z/TPF does not

support J2EE, much of what
is in these standard deployment descriptors does map to z/TPF SOAP support.

9 Roles

Additional roles will be taken on by existing TPF development and operations/coverage staff to
support the SOA development and production environments. Indeed, opportunities will exist to
enable nontraditional support staff, such as business analysts and marketing specialists, to
leverage new and existing services to create new business solutions. In an SOA environment,
development programmers will build services, use services, and develop solutions that aggregate
services. Operations/coverage personnel will manage services by performing deployment, publishing,
and activation activities on the z/TPF host.

You can refer to the table in Appendix A for a definition of roles in a traditional z/TPF data
processing environment. Appendix B contains a table listing a comprehensive set of roles that
pertain to an SOA data processing environment. These tables can be used to help determine how
best to broaden roles in an existing z/TPF data processing environment to realize the benefits
of an SOA environment. These tables will also assist in determining what new roles can be used
to develop business solutions using services available to an enterprise across heterogeneous hosts.

9.1 Development Roles

Traditional roles for the development programmer typically consist of creating artifacts such
as source code, runtime SOs, and documentation. The responsibilities of a development programmer
will expand to create new artifacts to define services for the runtime artifacts. New roles
may be created to leverage these services to create new business solutions using aggregation.
Roles to support development in an SOA environment can include:

Application Programmer, develops the business application and services according to the planned
architecture.

Business Customizer, customizes business application components and processes.

Component Developer, creates individual modules of software intended to be integrated into and
reused across multiple applications.

Integration Developer, creates new business functions by combining existing components.

9.2 Operations/Coverage Roles

Traditional roles for a z/TPF operator and coverage programmer are the deployment of applications on the z/TPF host, the monitoring of the host, and the day-to-day administration the host may require. New roles will need to be created to manage the configuring, operating, and monitoring of services that z/TPF will contain. Roles to support operations and coverage may include:

Coverage Programmer, is responsible for daily activities of the system and application operation; including system maintenance and software updates.

System Programmer, provides second-level support of the system and ensures system integrity by monitoring system resource usage. Also provides system support for hardware and system software upgrades.

Release Deployer, installs and deploys new or updated business solutions onto the host.

Asset Manager, identifies, collects, and maintains inventory of business assets throughout their lifecycle.

9.3 Other Roles

Other roles will exist that would not necessarily be part of the z/TPF development team, but would make use of services residing on the z/TPF host. These roles would leverage the services available to an enterprise across all of their hosts, regardless of the system type. Even services residing outside an enterprise’s realm can be used to develop a business solution (for example, a check clearing service).

Business Strategist, analyzes business issues and recommends solutions.

System Analyst, analyzes, evaluates, and designs systems to meet identified business requirements.

Software Architect, defines the architecture for a software application or component.

10 Tooling

Tooling is crucial to the consumability and, therefore, adoption of Web services. Consumability refers to the time and effort that it takes for customers to get to the point where they are actually using the functionality provided in the runtime. Said another way, if z/TPF provides Web services support in the runtime (or online system) without the requisite tooling support, z/TPF customers will be forced learn about HTTP, XML (namespaces, schema, …), SOAP, WSDL, UDDI, and the WS-I just for starters. At the time of this writing there are dozens of WS-* standards and profiles in development that extend, enrich, and complicate the entire Web services space. By pushing this responsibility to customers we are ensuring that, at best, the slow adoption of Web services, and, at worst, no adoption of Web services; further isolating and restricting the access to z/TPF in the enterprise.

At a high level, the main goal of adopting Web services is to easily integrate the business logic and/or data assets of one system with those on other heterogeneous platforms that may exist internally or externally to your organization/company. This means that the adoption of Web services does not entail the discarding of assets. On the contrary, businesses should, in the first phases of Web services adoption, examine the current assets that exist on their platforms and consider exposing portions of them via Web services interfaces. The benefit of doing this is that by exposing something as a Web service, it is available to all the various platforms that comprise an enterprise through one channel versus n channels. Subsequent phases of Web services deployment can include the decomposition of current assets into their components, some of which may be better addressed or implemented off of the current platform, and possibly by a third part/partner, thus allowing for platform optimization.

Figure 8: Enterprise Integration

As an architectural concept, SOA permits multiple approaches for the realization and deployment of an IT system that has been designed and built around its principles. In fact, there have been many enabling technologies for building SOAs over the years, including CORBA, J2EE, DCOM, and MQSeries. These technologies provided advancements at the time, but each has specific limitations.

From the technology side, the past 15 years have resulted in a realization of the importance of middleware standards and architectures, learning in particular from the successes

and failures of distributed object systems and message-oriented middleware.

One specific technology that arguably has the most significant commercial visibility and traction is Web services.

Web services describe a standardized way of constructing and integrating applications using XML based open standards over an Internet backbone. What makes the application of Web services as an enabling technology for SOA so powerful is that, for the first time, we have an underlying mechanism that uses well defined, standardized interfaces and “wire level” formats and protocols that facilitate interoperability and also effectively freeing the calling program from the need to deal with the intricacies of invoking the underlying services that comprise the applications – i.e. virtualising the application and its composite services. While there are other technologies that could provide a foundation for building and delivering SOA, the key value of Web services is its almost universal support across the IT industry.

If you take the driving factor mentioned of “freeing the calling program from the need to deal with the intricacies of invoking the underlying services”, and extend it to include the idea of freeing an application from having to deal with the intricacies of exposing itself as a service, the importance of tooling to complement the Web services runtime then becomes clear.

As mentioned throughout the Artifacts section of this document, tooling is a key point to the deployment of Web services. The goal of the tooling is to shield the user from having to know the particulars and details about the various Web services specifications (for example, WSDL, SOAP, UDDI, and XML). At the core of the tooling requirement is the capability of taking a description of a Web service (the WSDL description) and creating the code that can transform between the data encapsulated in the SOAP messages and native data structures that can be used by the applications that are operating on a particular runtime. Some runtimes have benefited from being viewed as the “runtime of the day”, like J2EE, from the standpoint of tooling support for Web services (for example, Java2WSDL, WSDL2Java, Java2Schema, and Schema2Java), but this ignores the large amount of “legacy” assets that exist in the enterprise today. What is needed at this point is a similar effort to create C/C++ based tooling that can generate code to work with C/C++ based runtimes.

This lack of tooling has not stalled the deployment of Web services for all of the C/C++ based runtimes, but it has resulted in each runtime having to create its own tooling. Moving toward a common set of C/C++ based tooling would accelerate the deployment of Web

services.

For further details, shown in the following figure, there is tooling in the J2EE space that
generates code based on a service's WSDL that would be analogous to the Web service drivers and
Web service stubs. The J2EE tooling also provides for the generation of WSDL from .class files.

Figure 9: Web services Data Transformation

Tooling Requirements:

The following section will briefly explain the tooling in relation to the Web services standards.

WSDL
• Top-down: Starting with a WSDL

o Provider: Starting from the WSDL, the tooling would generate the majority of a Web service driver,
and skeleton code for the application that is called by the Web service driver. The user is left to
fill in the business logic of the application.

o Consumer: Starting from the WSDL, the tooling would generate a Web service stub and a C header file
for the application to use.

• Bottom-up: Starting with an application

o Provider: Starting from the application, the tooling would help create the WSDL and generate the
majority of a Web service driver.

SOAP Messages
• Web service Drivers: This is code (generated by the tooling) to be installed into the runtime that
can take as input SOAP requests (XML Schema datatypes) and transform the data in them to datatypes
that are native to the runtime, and then take application return information and transform that from
the native datatypes of the runtime into SOAP response data. The format of these SOAP requests and
responses is defined by the WSDL for a Web service. These are for Provider-side deployments.

• Web service Stubs: This is code (generated by the tooling) to be installed into the runtime that
can take as input application request data and transform that from the native datatypes of the
runtime into SOAP request data (XML Schema datatypes), and then take the SOAP response data and
transform that to datatypes that are native to the runtime for the

requesting application to use.
The format of these SOAP requests and response is defined by the WSDL for the Web service. These
are for Consumer-side deployments.

UDDI
• At the time of this writing, UDDI support in the runtime for z/TPF is not foreseen, but this does
not mean that UDDI support in some fashion in the tooling is not needed. At a high level, UDDI can
be used to advertise Provider Web services that have been deployed to a specific runtime, thus
allowing potential Consumers to obtain the WSDL for these Web services. The tooling support for
UDDI should include the ability to publish WSDL files and any other required description files
(WSIL?), to a UDDI service registry.

11 Web Services Standards

Any paper discussing SOA and an implementation of SOA based on Web services would be incomplete
without a discussion of the various applicable standards and the future direction for the adoption
of these standards. Many new to the area of Web services, and even some who have been working in
the area for a while, find the proliferation of all the new, and often times conflicting WS-*
standards, confusing and overly complex. This section will attempt to provide the reader with a
higher level view of the set of Web services standards and describe the manner in which support
for them will be provided. To start, the Web services standards can be decomposed into three
categories: base standards, optional standards, and profiles.

Base Standards
The following is a list of what could be considered the Base standards in the area of Web services:
• WSDL 1.1

• SOAP 1.1/1.2

• UDDI 2

These provide for the find-bind-execute paradigm of Web services. The z/TPF Runtime currently
provides support for the Provider side of SOAP 1.1/1.2. SOAP consumer and intermediary support
in the z/TPF Runtime is currently not included in any committed plan, but is actively being
investigated for inclusion in a future product update.

There are currently no plans for the inclusion of WSDL and UDDI in the z/TPF Runtime. As mentioned in section 4.2.2, there are currently no plans to implement dynamic Consumer/Client support, which is mainly what WSDL and UDDI would be used for in the z/TPF Runtime. For WSDL and UDDI support in the TPF Toolkit, there are currently no committed plans, but the two are actively being investigated for inclusion in a future release. WSDL could be used in the z/TPF Runtime to provide for, say, SOAP message validation, but this is currently not in plan because the XML Scanner that is currently used in the runtime does not support it, and if it were possible with our current XML Scanner it would still not likely be supported due to the performance impacts of XML message validation based on an XML Schema.

Note that implicitly all of the Base standards build upon the standards around the eXtensible Markup Language (XML), as each of the above standards define languages that are XML based.

Optional Standards

As mentioned above, the Base standards provide for the basic find-bind-execute sequence. Those Base standards do this in a way that does not provide security, transactions, robust messaging, etc… The goal of the Optional standards is to address these other capabilities and requirements. As mentioned in Section 5, message handlers provide for this optionally processing at the SOAP layer, meaning that to implement a new Web services standard in the z/TPF Runtime, a new message handler would be developed that implements the particular standard.

The two main standards organizations involved in Web services are

- World Wide Web Consortium (W3C) at http://www.w3c.org
- Organization for the Advancement of Structured Information Standards (OASIS) at http://www.oasis-open.org

This site contains an exhaustive list of the current and proposed standards in the Web services area. There is no standard definition of the Web Services Protocol stack, though the W3C Web Services Architecture Working Group did publish a Web Services Architecture document, which provides a n excellent context for the various protocols. The following figure shows the various standards and categorizes them into domains:

Figure 10: Web Services Standards and Domains

Profiles

Although the basic premise of Web services and the role of the standards bodies are to achieve interoperability across platforms, operating systems, and programming languages, standards specifications are always open to interpretation. As a means to constrain the myriad permutations of Web service functional combinations, the WS-I organization has begun publishing “usage scenarios” and “profiles”, which document usage guidelines, required support, usage constraints, sample applications and test verification suites. These named profiles provide a simplified vocabulary for discussing Web Services, which helps producers and consumers of Web services technology to focus on understanding and addressing real needs for interoperability.

Other organizations, like industry-specific groups or companies, can be producers of profiles to address certain usage scenarios that may not be covered by other profiles, but need to be agreed upon for the efficient operation of an industry or group. It is expected that either such profiles would be adopted, and owned, by an organization like the WS-I or reach a level of maturity where they might be useful to other organizations or groups.

Based on this, z/TPF will focus on supporting standard profiles versus individually selecting from the WS-* set of standards unless driven by an explicit customer requirement. So, as can be seen below, we plan to implement and conform to the WS-I Basic Profile and the WS-I Simple SOAP Binding Profile, which are the profiles that address interoperability issues related to the Base SOAP, WSDL and UDDI, while using HTTP as the transport. The WS-I Basic Security Profile is the next Profile that will be supported based on the general concern of "we need security" from our customers, while not being able to explicitly define in a formal requirement what is meant by "secure Web services". Many actually view WS-I profiles as the mark of maturity in that a standard cannot be deemed mature until such time as there is a profile that describes how to use it in an interoperable way. A former Merrill Lynch CTO has been quoted as saying, “If you're an infrastructure player and don't buy into the WS-I group [Web Services Interoperability], don't even show up--we won't do business with you.”

The following list includes the Profiles that are not in a committed plan, but are being actively investigated for inclusion in a future z/TPF PUT release:

- WS-I Basic Profile
- WS-I Simple SOAP Binding Profile
- WS-I Basic Security Profile

The following two Profiles are emerging and becoming widely adopted. and z/TPF would consider
supporting them based on explicit customer need:

- WS-I Attachments Profile
- IBM Reliable Asynchronous Messaging Profile (RAMP)

12 z/TPF Architecture to Support SOA

12.1 Common SOAP Engine

The following diagram shows the proposed SOAP Engine with its interfaces which is being developed
in the Communications Infrastructure Workgroup of the Enterprise Software Architecture Board.
Note that the interfaces shown in this section are written such that they do not comply with z/TPF
API name standards for externalized APIs because this SOAP Engine is being developed in conjunction
with multiple teams across IBM. The z/TPF implementation of the Common SOAP Engine would include
mappings from these interfaces to the z/TPF-specific interfaces that conform to name standards.

12.1.1 SOAP Engine Interfaces

The SOAP engine consists of the eight categories of interfaces listed below:

- Consumer Interface
- Provider Interface
- Parser Interface
- Service Registry Interface
- Transport Interface
- Handler Interface
- Application/Wrapper/Stub Interface
- SOAP Fault Builder Interface

Some of the interfaces that are listed will contain a singular C API, while others may consist of a collection of C APIs. For interfaces that contain multiple APIs, the general approach of creating a handle-type structure will be used if necessary. The design of the handle structures will need to provide for extensibility. Each of the interfaces are expanded upon below.

Consumer Interface: The Consumer Interface provides access to service providers by applications located on the host with this SOAP engine. The following APIs will make use of the SOAPRequestHandle:

- BuildSOAPRequest()
- SetOptsSOAPRequest()
- SendSOAPRequest()
- ReceiveSOAPReply()

Provider Interface: The Provider Interface provides access the applications operating on the same host as the SOAP engine.

- HandleSOAPRequest(struct SOAPmsg *input, struct SOAPmsg *output, struct TransportHandle *comms)

Parser Interface: The Parser Interface provides the SOAP engine with access to a parser for parsing SOAP messages, for the creation of SOAP messages, and for data translation between XML schema datatypes and datatypes native to the operating environment. These APIs will make use of the XMLParserHandle.

Service Registry Interface: The Service Registry Interface provides for the ability to dynamically alter (not requiring code changes or recompiles) 1) which applications are deployed/available on the provider and consumer sides, 2) the message handlers that are required for a particular Web service (consumer or provider), 3) the transports allowed for particular Web services, and 4) the service wrappers/stubs for a particular Web service. The main interface for the service registry is a lookup interface that will search the registry for information about the requested Web service name. The SOAPreg structure will contain instructions to be used by the three portions of the SOAP Engine (for example, Core, Handler Engine, and Data Transformation) to correctly handle the Consumer or Provider request/response.

- LookUpSOAPRegistry(struct SOAPreg *regentry)

Transport Interface: The Transport Interface is used to provide the SOAP engine with the necessary information to handle SOAP Provider and Consumer requests. The following APIs will make use of the TransportHandle:

- InitTransportHandle()
- SetOptsTransport()
- OpenTransportConnection()
- CloseTransportConnection()
- GetTransportProperty()
- SetTransportProperty()
- GetTransportServiceName()
- TerminateTranspotHandle()

Handler Interface: The Handler Interface is meant to allow for the easy addition of message-level handlers that operate on, or create portions of, the SOAP message header. For example, there could be a message handler that implements WS-Security.

Application/Wrapper/Stub Interface: The Application/Wrapper/Stub Interface provides the interface between the SOAP engine and the Applications that are either the Providers or the Consumers, and they are responsible for any remaining data transformation that must be performed.

SOAP Fault Builder Interface: The SOAP Fault Builder Interface will create a well-formed SOAP fault message to be returned to the caller of either the Consumer or Provider Interface.

- BuildSOAPFault(struct SOAPfault *fault, struct SOAPmsg *output)

12.1.2 SOAP Engine Internals

12.1.2.1 Core

The Core of the Common SOAP Engine is responsible for implementing the SOAP specification, ensuring that the SOAP messages (requests and responses) conform to the applicable version of the SOAP specification. The SOAP Core will need to provide for the ability to specify which version of SOAP to conform to (that is, SOAP v1.2 or SOAP v1.1+WS-I Basic Profile v1.1).

12.1.2.2 Handler Engine

As mentioned in the Artifacts section, the SOAP specification provides an extensibility mechanism whereby optional functionality can be performed on SOAP messages. Because this functionality is optional, it is expected that deployers of Web services will want to pick and choose which optional functionality is required for a given Web service. The definition of a deployed Web service (either Consumer or Provider) contained in the service registry entry will contain a list of the message handlers that must be involved in the processing of request and responses messages. For example, if a Provider Web service is deployed, named MyService, requires WS-Security processing, which is not required for all messages flowing through the SOAP engine, the service registry entry for MyService will then include the WS-Security message handler name. This message handler name will then point to the installed implementation of the message handler, and the Handler engine will be responsible for ensuring that all messages for MyService flow through the WS-Security handler.

12.1.2.3 Data Transformation

The Data Transformation portion of the SOAP Engine is very similar to the Handler Engine. It is a mechanism that directs which functionality is performed on a particular SOAP message, and this functionality is responsible for the transformation between SOAP/XML datatypes and the native datatypes of the operating environment of the application. Like the Handler Engine, the definition of a deployed Web service contained in the service registry entry will contain either the name of the Web service driver (Provider Web service) or the Web service stub (Consumer Web service) that the data transformation portion of the SOAP engine should use to process SOAP messages.

12.2 Parser

Different parsers have different processing behavior and characteristics, and the SOAP Engine should not place requirements on the particular system to use a particular XML parser. The parser API will be used pervasively throughout the SOAP Engine. It will be used by the Core of the SOAP engine to perform SOAP m essage format validation, the individual message Handlers to implement the optional functionality for a particular Web service, and the Data Transformation (drivers/stubs) to perform the conversion between SOAP/XML datatypes and the native datatypes. This API will need to be defined and used by Tooling to support the SOAP Engine because it is this Tooling that will be generating the Web service drivers and

stubs.

12.3 Service Registry

The service registry is a mechanism that is accessed by the SOAP Engine to access definitions of the deployed Web services (Consumer and Provider) and Message Handlers in the system. On the Consumer side, an application will invoke a lookup request to the service registry for a given Web service. The service registry would return the definition of that Consumer Web service. The application will then call an API to set up parameter information and set any options for the request. Lastly, the application will call an invoke API to actually cause the SOAP request to be built and transferred to the Provider. The definition of the Web service returned by the service registry provides all of the information needed by the SOAP Core, Handler Engine, and Data Transformation component. On the Provider side, a request that arrives at the SOAP Engine via the Provider interface will result in a lookup request to the service registry for the requested Web service, which is a parameter on the Provider API. Like the case for the Consumer side, the returned definition is then used by the SOAP Core, Handler Engine, and Data Transformation component to provide for the requested Web service.

12.4 Transports

12.4.1 HTTP

Apache 1.3 is the supported transport for HTTP.

12.4.2 HTTPS

The TPF statement of direction is to support HTTPS using Apache 1.3 + mod_ssl.

12.4.3 MQSeries

12.4.3.1 MQ Bridge

z/TPF SOAP server support can be used with WebSphere MQ. In this case, a user-written Websphere MQ monitor that receives a message can call the tpf_soap_handler() directly. The monitor must set up the applRoutingInfo string in the commsBinding structure (defined in the c_soap.h header file) that is used for routing the SOAP request to the appropriate z/TPF application. The applRoutingInfo string can be set up in many ways; for example, it can be established by a one-to-one mapping between a queue name and an

applRoutingInfo string, or by defining the MQ message structure to contain both the SOAP message and the applRoutingInfo information. Regardless of the approach, the tpf_soap_appl_handler() user exit must be updated, similar to how it is updated for SOAP over HTTP requests, to access the z/TPF application.

12.5 Message Handler

The Message Handlers will be modules (executable code) loaded to the system that implement some functionality that is not in the base SOAP specification, but rather is optional functionality from the standpoint that not every message that flows through the SOAP Engine may be required to use it. Each Message Handler will have to implement two interfaces: one for processing a SOAP Request message (either Consumer or Provider side) and one for processing a SOAP Response message (either Consumer or Provider side). A Message Handler is deployed to the SOAP Engine in a similar manner to how Web services are deployed to the SOAP Engine by way of a deployment descriptor. The deployment descriptor is an XML file that conforms to the Message Handler XML Schema (provided by this support), and it provides all of the necessary information needed by the Handler Engine to make use of the particular Message Handler.

12.6 Service Wrapper

The Service Wrappers map one to one to the Web services that are deployed (as Providers) to the SOAP Engine. These will be modules loaded to the system that provide the following functionality for a particular Web service:

- Transform the SOAP Body/XML parameter data from the request into the native datatypes for use by the application that implements the Web service
- Invoke the appropriate application with the parameter data
- Handle the response from that application and transform the native datatype returned by the application into the SOAP Body/XML for the response.

When a Provider Web service is deployed to the SOAP Engine, its deployment descriptor, which conforms to the Web service deployment descriptor XML Schema (provided by this support), will reference the Service Wrapper that is to be used to satisfy SOAP requests.

12.7 Service Stub

The Service Stubs map one to one to the Web services that are deployed (as Consumers) to the SOAP Engine. These will be modules loaded to the system that provide the following functionality for a particular Web service:

• Transform the native datatype parameter data for the request into the SOAP Body/XML for the request

• Invoke the SOAP Engine to complete the creation of the SOAP request and send it to the Provider

• Handle the response from Provider, via the SOAP Engine, and transform the SOAP Body/XML response
information into the native datatypes and return this to the application.

When a Consumer Web service is deployed to the SOAP Engine, its deployment descriptor, which conforms to
the Web service deployment descriptor XML Schema (provided by this support), will reference the Service
Stub that is to be used to satisfy SOAP requests.

12.8 Application

There are three ways that an application can participate in a z/TPF SOA environment: as a service provider
by using the Service Wrapper to interface with the SOAP Engine, as a service consumer by using the Service
Stub to communicate a service request to the SOAP Engine (these are not mutually exclusive), and by direct
connection to the SOAP Engine and being responsible for manipulating the SOAP/XML message and formatting
the response SOAP/XML message itself.

Appendix A Web Services Example

This example will show a mythical airlines availability transaction and highlight two patterns of enabling
the transaction as a Web service. The first pattern will show how Wrapping can be used to expose a transaction
as a Web service, and the second pattern will use Refacing to expose the transaction. Artifacts that are
common to both patterns will be shown first, as well as the message format of our mythical transaction and
other prerequisite information.

Transaction Criterion

This transaction will consist of a primary action code, ‘5’, indicating an availability request, followed
by a date in the format of DDMMM and then a city-pair. We will use the following message in our examples:

```
517JULJFKLAX
```

This transaction is initially processed by segment UII1 and its primary action code indicates that the
message is to be processed by TXA0. The availability package will

attempt to find up to 5 flights that
match the criterion, and each iteration will enter FMSG to build the
response and then enter FMSG a final
time indicating that the response message is to be sent.

Deployment Descriptor

A new Web services driver will be written (WXA0) that will handle the
new XML/SOAP interfaces. The function
it delivers will differ between the two patterns. These differences will
be shown in their respective
sections. The following is the deployment descriptor that will be used
to implement the driver:

```
<?xml version="1.0" encoding="UTF-8"?>
<WebService xmlns="http://www.ibm.com/tpf"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.ibm.com/tpf/webservices/tpfWebService.xsd
">
 <WebServiceName>Availability</WebServiceName>
      <ContextPath>/pars/webservices/</ContextPath>
 <DriverName>WXA0</DriverName>
 <ImplementationLanguage>Assembly</webServiceImplementationLanguage>
 <WSICompliance>Yes</wsiCompliance>
 <ActivationType>Automatic</activationType>
 <Operation>getPARSService</Operation>
 <Operation>getAvailability</Operation>
</webService>
```

Wrapping Pattern

This pattern simply exposes the interface to the application. It allows
for increased access but is
limited in the reuse and flexibility capabilities of other patterns.
The advantage of this pattern
is that it requires minimal changes to the application.

WSDL

The XML/SOAP message described by the WSDL is essentially a
“screenscrape” of the existing request
and response messages. The consumer of the service must be aware of the
esoteric message format. The
following is an example of the WSDL to support this pattern:

```
<?xml version="1.0" encoding="UTF-8" ?>
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://tempuri.org/PARSService/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="PARSService"
targetNamespace="http://tempuri.org/PARSService/">
<wsdl:types>
<xsd:schema targetNamespace="http://tempuri.org/PARSService/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:complexType name="getPARSService">
<xsd:sequence>
  <xsd:element name="ScreenScrapeReq" type="xsd:string" />
  </xsd:sequence>
```

```
  </xsd:complexType>
<xsd:complexType name="getPARSServiceResponse">
<xsd:sequence>
  <xsd:element name="ScreenScrapeRes" type="xsd:string" />
  </xsd:sequence>
  </xsd:complexType>
  <xsd:element name="PARSServiceRequest" type="tns:getPARSService" />
  <xsd:element name="PARSServiceResponse"
type="tns:getPARSServiceResponse" />
  </xsd:schema>
  </wsdl:types>
<wsdl:message name="PARSServiceResponse">
  <wsdl:part name="PARSServiceResponse"
element="tns:PARSServiceResponse" />
  </wsdl:message>
<wsdl:message name="PARSServiceRequest">
  <wsdl:part name="PARSServiceRequest"
element="tns:PARSServiceRequest" />
  </wsdl:message>
<wsdl:portType name="PARSService">
<wsdl:operation name="getPARSService">
  <wsdl:input message="tns:PARSServiceRequest" />
  <wsdl:output message="tns:PARSServiceResponse" />
  </wsdl:operation>
  </wsdl:portType>
<wsdl:binding name="PARSServiceSOAP" type="tns:PARSService">
  <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http" />
<wsdl:operation name="getPARSService">
  <soap:operation
soapAction="http://www.tpfsystem.com/WebServices/PARSService/"
style="document" />
<wsdl:input>
  <soap:body use="literal" />
  </wsdl:input>
<wsdl:output>
  <soap:body use="literal" />
  </wsdl:output>
  </wsdl:operation>
  </wsdl:binding>
<wsdl:service name="PARSService">
<wsdl:port binding="tns:PARSServiceSOAP" name="PARSServiceSOAP">
  <soap:address location="http://www.tpfsystem.com/WebServices" />
  </wsdl:port>
  </wsdl:service>
  </wsdl:definitions>
```

Driver

The Web services driver program receives control from the SOAP engine when an availability request is
received (as specified by the deployment descriptor). The driver will perform validations of the request
and report anomalies via the SOAPFault interface. When the request message has been validated, a core
block is retrieved on level 0 and preformatted to appear as an MI0MI z/TPF standard input message.

XML element ScreenScrapeReq is retrieved from the XML/SOAP request message. The string in this element

is then copied to the MI0MI core block and the message length is updated. An indicator is set in the ECB
to mark this as a Web services transaction. Control is then sent via an ENTRC macro to the initial
availability segment, TXA0.

Application

The application changes are simply to check whether this ECB is a Web services ECB before entering FMSG
for the final time. If this is a Web services ECB, a BACKC macro is issued instead to allow the driver to
gain control and format the XML/SOAP response.

Driver

When control is returned to this driver, a new XML/SOAP document is created to form the response message.
The OM0SG z/TPF standard output message is retrieved from level 6 and the data that would have been
displayed on the agent screen is instead added to the ScreenScrapeRes XML element. Chained messages will
need to be handled when applicable. The XML/SOAP response message is now complete, and the driver will
return to the SOAP engine to transmit the response back to the service consumer.

Sample Request Message

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://www.w3.org/2002/06/soap-envelope"
  SOAP-ENV:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
<SOAP-ENV:Body>
<ns1:getPARSService xmlns:ns1
="http://www.tpfsystem.com/WebServices/PARSService">
 <ScreenScrapeReq>17JULJFKLAX</ScreenScrapeReq>
</ns1:getPARSService>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Sample Response Message

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://www.w3.org/2002/06/soap-envelope"
  SOAP-ENV:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
<SOAP-ENV:Body>
<r:getPARSServiceResponse
xmlns:r="http://www.tpfsystem.com/WebServices/PARSService">
  <ScreenScrapeRes>
 AB 1023 1100 17JUL JFKLAX Y1 245.00&#0010;
 BC 99  1230 17JUL JFKORD Y1 215.00&#0010;
 BC 109  1415 17JUL ORDLAX S4&#0010;
 QQ 745  1500 17JUL JFKLAX Y2 275.00+
  </ScreenScrapeRes>
</r:getPARSServiceResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Refacing Pattern

The Refacing pattern would entail a minor refactoring effort that will separate the business logic
from the message interface. In this way the original access to the application would remain intact
and the Web services interface can be added. (Indeed, other methods of interface can be added as
well.) This pattern provides for increased access and facilitates usage of the service by a consumer.
Greater reuse and flexibility are realized with this pattern.

WSDL

The XMP/SOAP message described by the WSDL defines each individual field necessary for the availability
service to process the request. This is equally true for the response message. This facilitates the
communication between the consumer and provider by removing the necessity of knowledge of esoteric message
formats. The following is an example of the WSDL to support this pattern:

```
<?xml version="1.0" encoding="UTF-8" ?>
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://tempuri.org/AvailabilityService/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="AvailabilityService"
targetNamespace="http://tempuri.org/AvailabilityService/">
<wsdl:types>
<xsd:schema targetNamespace="http://tempuri.org/AvailabilityService/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:complexType name="CityPair">
<xsd:sequence>
  <xsd:element name="DepartureCity" type="xsd:string" />
  <xsd:element name="DestinationCity" type="xsd:string" />
  </xsd:sequence>
  <xsd:attribute name="non-stop" type="xsd:string" use="optional" />
  </xsd:complexType>
<xsd:complexType name="Date">
<xsd:sequence>
  <xsd:element name="Day" type="xsd:string" />
  <xsd:element name="Month" type="xsd:string" />
  </xsd:sequence>
  </xsd:complexType>
<xsd:complexType name="getAvailability">
<xsd:sequence>
  <xsd:element name="Date" type="tns:Date" />
  <xsd:element name="CityPair" type="tns:CityPair" />
  </xsd:sequence>
  </xsd:complexType>
<xsd:complexType name="getAvailabilityResponse">
<xsd:sequence>
  <xsd:element name="carrier" type="xsd:string" />
  <xsd:element name="flight" type="xsd:string" />

  <xsd:element name="time" type="xsd:string" />
  <xsd:element name="Date" type="tns:Date" />
```

```
  <xsd:element name="CityPair" type="tns:CityPair" />
  <xsd:element name="class-of-service" type="xsd:string" />
  <xsd:element name="price" type="xsd:string" />
  </xsd:sequence>
  </xsd:complexType>
  <xsd:element name="availabilityRequest" type="tns:getAvailability" />
  <xsd:element name="availabilityResponse"
type="tns:getAvailabilityResponse" />
  </xsd:schema>
  </wsdl:types>
<wsdl:message name="availabilityResponse">
  <wsdl:part name="availabilityResponse"
element="tns:availabilityResponse" />
  </wsdl:message>
<wsdl:message name="availabilityRequest">
  <wsdl:part name="availabilityRequest"
element="tns:availabilityRequest" />
  </wsdl:message>
<wsdl:portType name="AvailabilityService">
<wsdl:operation name="getAvailability">
  <wsdl:input message="tns:availabilityRequest" />
  <wsdl:output message="tns:availabilityResponse" />
  </wsdl:operation>
  </wsdl:portType>
<wsdl:binding name="AvailabilityServiceSOAP"
type="tns:AvailabilityService">
  <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http" />
<wsdl:operation name="getAvailability">
  <soap:operation
soapAction="http://www.tpfsystem.com/WebServices/AvailabilityService/"
style="document" />
<wsdl:input>
  <soap:body use="literal" />
  </wsdl:input>
<wsdl:output>
  <soap:body use="literal" />
  </wsdl:output>
  </wsdl:operation>
  </wsdl:binding>
<wsdl:service name="AvailabilityService">
<wsdl:port binding="tns:AvailabilityServiceSOAP"
name="AvailabilityServiceSOAP">
  <soap:address location="http://www.tpfsystem.com/WebServices" />
  </wsdl:port>
  </wsdl:service>
  </wsdl:definitions>
```

Driver

The Web services driver program receives control from the SOAP engine when an availability request
is received (as specified by the deployment descriptor). The driver will perform validations of the
request and report anomalies via the SOAPFault interface. When the request message has been validated,
the individual fields in the XML/SOAP request message that are pertinent to the availability message
are extracted and placed in a DSECT or C struct that is defined for this purpose. Control is then

sent via an ENTRC macro to the initial availability segment, TXA0.

Application

The application is refactored to separate the business logic from the message interface. References to
the input data are now made to a DSECT/C struct instead of direct reference to the MI0MI core block.
Conversely, output is now written to a DSECT/C struct to contain the response array. When processing
is complete, a BACKC macro is executed to return to the appropriate driver that will format the response
DSECT/C struct block into the format expected by the requester/consumer.

Driver

When control is returned to this driver, a new XML/SOAP document is created to form the response message.
The DSECT/C struct that is defined to contain the response data is used to extract the information that
is to be returned to the consumer. This data is added to the XML elements in the response message. The
XML/SOAP response message is now complete, and the driver will return to the SOAP engine to transmit the
response back to the service consumer.

Sample Request Message

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://www.w3.org/2002/06/soap-envelope"
  SOAP-ENV:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
<SOAP-ENV:Body>
<ns1:getAvailability xmlns:ns1
="http://www.tpfsystem.com/WebServices/AvailabilityService">
  <Date>
    <Day>17</Day>
    <Month>JUL</Month>
  </Date>
  <CityPair>
 <DepartureCity>JFK</DepartureCity>
 <DestinationCity>LAX</DestinationCity>
  </CityPair>
</ns1:getAvailability>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Sample Response Message

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://www.w3.org/2002/06/soap-envelope"
  SOAP-ENV:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
<SOAP-ENV:Body>
<r:getAvailabilityResponse
xmlns:r="http://www.tpfsystem.com/WebServices/AvailabilityService">
 <carrier>AB</carrier>
 <flight>1023</flight>
 <time>1100</time>
```

```
 <Date>
     <Day>17</Day>
     <Month>JUL</Month>
 </Date>
 <CityPair>
  <DepartureCity>JFK</DepartureCity>
  <DestinationCity>LAX</DestinationCity>
   </CityPair>
 <class-of-service>Y1</class-of-service>
 <price>245.00</price>
 <carrier>BB</carrier>
 <flight>99</flight>
 <time>1230</time>
 <Date>
     <Day>17</Day>
     <Month>JUL</Month>
 </Date>
 <CityPair>
  <DepartureCity>JFK</DepartureCity>
  <DestinationCity>ORD</DestinationCity>
   </CityPair>
 <class-of-service>Y1</class-of-service>
 <price>215.00</price>
 <carrier>BB</carrier>
 <flight>109</flight>
 <time>1415</time>
 <Date>
     <Day>17</Day>
     <Month>JUL</Month>
 </Date>
 <CityPair>
  <DepartureCity>ORD</DepartureCity>
  <DestinationCity>LAX</DestinationCity>
   </CityPair>
 <class-of-service>S4</class-of-service>
 <price></price>
 <carrier>QQ</carrier>
 <flight>745</flight>
 <time>1500</time>
 <Date>
     <Day>17</Day>
     <Month>JUL</Month>
 </Date>
 <CityPair>
  <DepartureCity>JFK</DepartureCity>
  <DestinationCity>LAX</DestinationCity>
   </CityPair>
 <class-of-service>Y2</class-of-service>
 <price>275.00</price>
</r:getAvailabilityResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Appendix B Web Services Scenarios

This section presents three scenarios of Web services transactions. They give the reader a feel for

how transactions in the SOA environment are constructed; more importantly, how the z/TPF system participates in this architecture. As the astute reader will see, the role of the Web-enabled z/TPF application in the SOA environment is not unlike its role in traditional z/TPF environments; z/TPF's strengths are (and historically have been): ultra-high availability, super-fast transactions, and massive databases. These characteristics play well for z/TPF to perform as the 'master' server that lies at the heart of an enterprise's data center.

Scenario Number One

In this scenario, a customer planning a weekend getaway is looking for inexpensive seats to her favorite destination:

1. Sally, from her home computer, connects to her favorite purveyor of inexpensive airline seats. She requests availability to her weekend destination.

2. CheepSeats.com sends queries to a list of airlines that fly from Sally's hometown to her destination. In our example, Zzyzx Airways is the last airline that is queried.

3. Zzyzx Airways has a front-end Web server (AirlineBroker.com) that performs as the gateway to their system of record. The front-end builds a SOAP message to send to their z/TPF system, where all of their reservation data is kept.

4. The Zzyzx z/TPF system processes the SOAP message and routes the request to the Web-enabled Availability application. The Availability application processes the request and sends the response back to the requester.

5. The response is forwarded to CheepSeats.com and then to Sally's Firefox Web browser. Now she is free to plan the rest of her getaway.

Scenario Number Two

In this scenario, a customer uses a bank's phone system to query account balances and make funds transfers:

0. Now that Harry, Sally's husband, took over the home computer system to play online poker, Sally calls

her bank’s automated phone system to make sure she has enough funds in her checking account so she can go shopping during her weekend getaway.

1. Sally first uses her bank’s automated phone system to query the balances for her checking and savings accounts.

2. The Ajax Bank phone system formats a SOAP message with the account query information and sends it to their z/TPF system, where the Account Balance application is located.

3. The Ajax Bank’s z/TPF system processes the SOAP message and routes the request to the Web-enabled Account Balance application. The Account Balance application processes the request and sends the response back to the requester.

4. The Ajax Bank phone system processes the response and presents the results back to Sally.

5. Sally decides to transfer $1,500 from her savings account to her checking account. She indicates her request to the Ajax Bank phone system.

6. The bank’s phone system formats the request into a SOAP message with the transfer information and sends it to the Account Transfer application, which happens to reside on the Ajax Bank’s z/TPF system.

7. The Ajax Bank’s z/TPF system processes the SOAP message and routes the request to the Web-enabled Account Transfer application. The Account Transfer application processes the request and sends the response back to the requester.

8. The Ajax Bank phone system processes the response and presents the results back to Sally.

9. Sally completed her banking transactions and hangs-up from the Ajax Bank automated phone system.

Scenario Number Three

In this scenario, a customer purchases an airline itinerary and car rental for a weekend getaway:

1. Sally, back on her home computer, reconnects to CheepSeats.com to purchase her airline tickets, as well as a car rental.

2. CheepSeats.com takes Sally's request and formats a SOAP message and sends it to the Zzyzx Airways internet gateway.

3. The Zzyzx Airways gateway finds the type of SOAP request and routes the request to the airline's web application server system.

4. The Zzyzx Airways application server finds the type of SOAP request and routes the request to the airline's Booking application, residing on the Zzyzx z/TPF system.

5. As part of its processing, the Booking application builds a SOAP request to send to the Fare Quote application to calculate the fare for the itinerary. The Fare Quote application processes the request and formats a SOAP message to respond to the requester.

6. The Booking application now formats a SOAP request to send to the Hominy Car Rental Company in order to book the car rental. Hominy's car rental reservation system processes the SOAP message and formats a SOAP response back to Zzyzx.

7. The Booking application completes the processing and formats a SOAP response with confirmation information back to the requester.

8. The response is forwarded back to CheepSeats.com and then to Sally's Web browser.

Scenario Number Three: Internal z/TPF Detail

Appendix C Traditional z/TPF Development Roles

The following are the roles found in a typical z/TPF development environment. These roles will continue in an SOA development environment and in many cases will have their responsibilities grow to support the SOA roles.

System Operations

Operations
Manages the day-to-day operation of the enterprise IT environment.

Operations Manager
Overall operations management; oversees all computer systems in an enterprise’s IT department

Operator
Monitors one or more specific operating system hosts

Coverage Manager
Provides overall leadership for all operation system and application integrity

Coverage Programmer
Assumes responsibility for the daily activities of system and application operation; includes system maintenance, software upgrades, and problem diagnosis

System Manager
Provides overall leadership for all z/TPF system integrity

System Programmer
Assumes responsibility for the daily activities of z/TPF system operation; includes system maintenance, software upgrades, and problem diagnosis

Test

Test Team
Overall responsibility for the design, implementation, and evaluation for all modifications scheduled for the production z/TPF system

QA Manager
Provides overall leadership for all z/TPF system integrity for the production system

QA Tester
Assumes responsibility for the daily activities of the test effort including test implementation, execution, validation, and diagnosis

SA Manager
Provides overall leadership for integration for application projects into a production environment

SA Tester
Assumes responsibility for the daily activities of the test effort including test implementation, execution, validation, and diagnosis

Development

Developer
Implements the business applications and services (e.g., data components, information, and required development tools) according to the architecture model

Application Manager
Provides overall people and resource management (and sometimes project management) to drive
business requirements

Application Programmer
Develops the business applications and services according to the planned architecture, incorporating
both functional as well as presentation aspects of the offering

Appendix D Web Service Project Roles

The following are some additional roles in a Service Oriented Architecture development environment
as presented in a DeveloperWorks article. This document can be reviewed at

http://www-128.ibm.com/developerworks/webservices/library/ws-roles/

The actual roles defined are at the discretion of the enterprise; more than one role can be assigned
to a single person and not every role needs to be filled. No roles are specific to z/TPF that cannot
fit into one of these role categories.

Extended roles

Product Vendor
Supplies a WS-I-compliant Web services runtime container, and optional service registry and SOAP
gateway services.

Deployer
Takes the development artifacts and installs them in the target runtime environment. Generates stubs
and skeletons for the target environment from WSDL and installs them together with the service implementations.

Tester
In charge of the various standard test stages such as integration, load, and acceptance test. Also
defines test cases for Web services interoperability and conformance tests.

Toolsmith
Designs and implements project-specific scripts, generators, and other utilities. The degree of standardization in the Web services world makes it possible to, for example, develop custom WSDL-, JAX-RPC- or JSR-109-aware tools.

Knowledge Transfer Facilitator
Provides access to subject matter experts and technical instructors who bring in extended knowledge regarding Web services concepts and implementation assets.

Extra roles

SOA Architect
Responsible for the end-to-end service requester and provider design. Takes care of inquiring on and stating the non-functional service requirements.

Service Modeler
Applies data and function modeling techniques to define the service interface contracts, including the shemas of exchanged messages.

Process Flow Designer
Investigates explicit, declarative service orchestration (aggregation, composition) possibilities. An optional role.

Service Developer
J2EE developer familiar with Web services concepts and XML. Develops service interface and implementation (provider side) and service invocation code (requester side).

Interoperability Tester
Verifies that the developed requester and provider implementations interoperate seamlessly and ensures Web Services Interoperability (WS-I) conformance.

UDDI Administrator
Defines how the generic UDDI data model is customized and populated. An optional role.

NOTICES

IBM may not offer the products, services, or features discussed in this information in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service. IBM may have patents or pending patent applications covering subject matter described in this information. The furnishing of this information does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Department 830A
Mail Drop P131
2455 South Road
Poughkeepsie, NY 12601-5400
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee. Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

TRADEMARKS

The following terms are trademarks of the International Business Machines Corporation in the United States, or other countries, or both:

IBM
CICS
DB2
IMS
MQSeries
WebSphere