An Approach to Flexible Application Composition in a Diverse Software Landscape

Roy Oberhauser

Corporate Technology, Siemens AG,
Otto-Hahn-Ring 6, 81730 Munich, Germany
roy.oberhauser@siemens.com

Abstract. With the escalating complexity, aggregation, and integration of software in enterprise, mobile, and pervasive arenas, it becomes increasingly difficult to compose, deploy, and operate applications that span a distributed and diverse software landscape. Furthermore, the increasing aggregation of software artifacts, including platforms, frameworks, components, services, and tools, lack a standard metadata description capability that hinders rapid and flexible distribution, deployment, and operation. This paper presents a general approach, realized with the FAST Framework, to improving the development, deployment, and operation of distributed applications that consist of diverse software artifacts. Application specification and composition is based on configuration queries that flexibly combine modules and a container that non-intrusively manages module lifecycles. The results show benefits with regard to simplified configurability, enhanced reuse via XML-based description propagation, improved distributed-application-provisioning intervals vs. local configurations, as well as applicability to Grid, Web Services, and MDA.

1 Introduction

As a trend, the complexity of software applications is escalating, where complexity is a function of the types and number of relationships among the software elements. This escalation is especially true of distributed applications in areas such as enterprise and pervasive infrastructures and applications. Increasing software integration as well as the aggregation of software artifacts, e.g. as shown in the tendency to utilize standardized platforms and API providers (e.g. J2EE, .NET), open source software frameworks, etc., contribute to the overall underlying complexity of an application.

Simultaneously, competitive pressures compel developers to more rapidly produce software that is parameterized to fit various predefined and hard-to-predict post-defined operational contexts and configurations. As these pressures in turn cause developers and maintainers to handle multiple projects simultaneously, when considered in conjunction with geographically distributed teams, the rapid reuse and propagation of deployment configurations will become a growing necessity.

Conversely, software operators (a set that includes developers) are faced with a daunting set of amassed choices with regard to both the parameterization and (reproducible) configuration of aggregated, distributed, and legacy software. This is exac-

erbated by the inherent variability of software, as in versioning of any of the constituent parts of a distributed application. This has given rise to the adage "Never touch a running system," and the problems in this area are described with case studies in [1]. While a myriad of configuration and deployment mechanisms exists, no unifying, widely adopted, practical, and economic solution in this diverse, heterogeneous software landscape is available.

Considering these aforementioned challenges, the primary objective is to support distributed application configurations (especially with regard to composability, flexibility, manageability, and reuse) by means of a non-intrusive infrastructure with minimal requirements for describing and provisioning the involved software artifacts. An artifact can be a piece of software or anything associated with it (e.g., tools, documentation). A solution that comprehensively addresses all possible software configurations and artifacts is beyond scope; rather, the contribution of this paper is a practical and economic approach that deals with various basic issues in the current gap in distribution, deployment, and operation, thereby drawing attention to this area.

The paper is organized as follows: Section 2 reviews related work to elucidate the current gap. In Section 3, the solution approach and constraints are presented. This is followed in Section 4 by a description of the solution realization, referred to in this paper as the FAST Framework. At the core of the framework is a container that manages modules and configurations and communicates with other containers. Section 5 discusses the results, and in Section 6, a conclusion is drawn.

2 Related Work

Because the focus of this paper touches such a universal aspect of software, it can easily be related in some way to a range of other efforts and work. Since not all related work and aspects can necessarily be addressed, only primary and/or key comparable work will be mentioned.

In the area of composition and integration, the VCF [2] approach to component composition relies on a Java API and requires component model plugins. Its focus is limited to components, and thus cannot provide a unifying approach that includes preexisting and non-component-oriented artifacts. FUJABA [3], a meta-model approach to tool integration, requires plug-ins and does not address distributed applications. [4] provides a survey of various composition environments, all of which cannot provide a unifying approach due to constraints, platform-dependencies, or a composition focus at the more intrusive communication or interaction level, resulting in development phase dependencies or runtime impacts. As to composability in Web Services (e.g., BPEL4WS, WSCI), its focus is the interaction abstraction level, leaving the infrastructural aspects of aggregation, configurability, distribution, provisioning, deployment, and manageability for a diverse software environment unaddressed.

As to platform-specific provisioning frameworks, SmartFrog [5] is a flexible, object-oriented framework for the deployment and configuration of remote Java objects. It has a security concept and its own declarative non-XML language includes parameterized and inheritable templates. However, its component model requires inheritance from Java objects - thus requiring a Java wrapper class for each external soft-

ware artifact it manages, and the more flexible concept of queries for application composition does not appear to be supported. Jini's Rio [6] provides dynamic provisioning capabilities, policies, and telemetry using an agent-based Dynamic Container. Service components are described using XML-based metadata (an Operational-String). A component model (Jini Service Beans) is provided and, for managing external services, Service Control Adapters must be written. Its model is platform-dependent and its reliance on Jini's RMI mechanism makes infrastructural interoperability in diverse landscapes problematic.

Common configuration management tools, such as HP's OpenView, IBM's Tivoli, etc., address various enterprise deployment and operation issues, yet they do not necessarily scale down for small projects, devices, or budgets, and various other tools are often tied to operating systems. Management standards activities, such as the Open Management Interface and OASIS Web Services Distributed Managmement Technical Committee, address management interfaces for Web Services (WS), but will likely not be able to manage all related and legacy artifacts. Platform-specific management initiatives, such as Java Management Extensions (JMX), will necessarily be limited to their platform, and have typically neglected the aspect of metadata.

Metadata initiatives, such as the WS-MetadataExchange, may allow WS to exchange metadata with one another, but first the WS must be operational, and no metadata standard exists for describing basic operational aspects of software artifacts. Platform-specific metadata such as contained in Java JAR files, in .NET, and in JSR 175 "A Metadata Facility for the Java Programming Language," are not easily accessible or modifiable by operators, especially for diverse environments. Where tooling creates the metadata, it tends to become so complex and all encompassing that manual creation for incorporation of other artifacts is avoided.

Model-Driven Architecture (MDA) [7] is primarily a development-centric paradigm and depends on the existence of models for the involved artifacts. While [8][9][10] describe the combination of Model-Integrated Computing (MIC) with middleware, the approach to distributed configuration and parameterization appears to be application synthesis, close integration with the CCM container CIAO, and in the future advanced meta-programming with reflection [11] and AOP [12]. How these distributed applications and each (non-modeled-)artifact are managed and configured in each operational environment variant over their lifetime, in conjunction with potential overlapping integration that occurs in SOA environments, is not detailed. While beyond the scope of this paper, in a model-centric environment a unifying approach for artifact metadata could conceptually, via XML Metadata Interchange (XMI) and the Meta-Object Facility (MOF), be integrated into such a tool chain to address various deployment, configuration, and parameterization aspects for artifacts at development-, initialization-, and run-time.

While viewpoints differ, Grid initiatives such as the Globus Alliance are primarily focused on resource sharing, security, performance, QoS, and the larger provisioning problem [13] with runtime interoperability protocols and toolkit APIs in view. Many (legacy) software artifacts are and will remain outside of this scope, yet the Grid resources being shared, e.g. via WSRF [14], may well be dependent on these hidden artifacts and their proper distributed configuration, management, and parameterization. A unified and cost efficient approach to these challenges is needed.

3 Solution Approach

To discuss the solution approach, it is helpful to work with an example that also illustrates the various challenges when different inter-related and inter-dependent software components, services, applications, frameworks, and infrastructures are combined. Although this illustration was realized for test purposes, the intent of this illustration is not to show an ideal distributed application, but rather to bring to the forefront the issues diverse artifacts and infrastructures can create.

As to solution constraints, no interference in the interfaces, installation, communication, and interactions between or within a module should necessarily occur, nor should changes be required to a module. Thus, encapsulation is not to be enforced, but rather a mechanism for developers to manually supply the relevant and missing metadata is provided, which may be partial but sufficient for the operational tasks. The relative simplicity and ubiquity of XML is a desirable quality to further adoption and standardization efforts, supports the rapid creation of missing module descriptions, and avoids coupling to the artifact itself (in contrast to JMX). Reuse of specified configurations and module descriptions as well as a lifecycle management mechanism shall be supported. Emphasis should be given to enhanced support for an operational view, benefiting operators, developers, and testers. Due to a lack of implementations and tools to realize this approach, a framework is necessary to interpret the configuration and module descriptions and manage the modules.

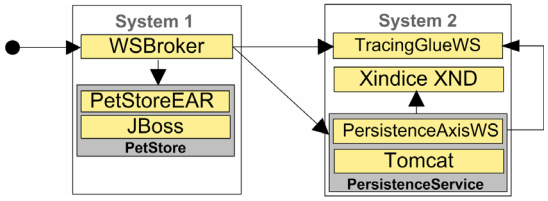

Fig. 1. Problem view showing sample distributed application interactions using diverse software artifacts

In Fig. 1 the problem view is presented using an illustration. Given a JBoss-ported reference J2EE PetStore enterprise application (PetStoreEAR) that is dependent on a J2EE application server JBossto become operational, this grouping of software artifacts can be considered a configuration PetStore. Another grouping, called the PersistenceService, consists of an XML persistence Web Service (PersistenceAxisWS) that abstracts XML Native Database (XND) differences, is based on the Apache Axis WS framework, is deployed as a web application (WAR) within a web server (Apache Tomcat), and uses Apache Xindice as an XND. Now hypothetically, without modifying either of these configurations, a PetStoreSupplier distributed application configuration would like to intercept all placed orders to the PetStore configuration, e.g. via an HTTP Proxy with interception capability (called WSBroker), persist these orders as XML via the PersistenceService configuration, and provide tracing data to a tracing Web Service implemented with the webmethods GLUE WS toolkit (TracingGlueWS). Note that the problem space could involve non-Java software.

To converge on a solution, the problem domain was partitioned into three separate areas:

1. The first area dealt with module description: what are the common attributes of software artifacts or modules that should typically be described, how should they be described, and what strategies can be used to reduce redundancy, manage change, and keep the involved effort and maintenance low?
2. The second area dealt with configuration description: how should modules best be flexibly aggregated into a grouping, how can redundancy be reduced, and how can the reuse or propagation of configurations be supported?
3. The third area consisted of the distributed software infrastructure necessary for supporting the lifecycle of modules and configurations with minimal intrusion and constraints in a diverse software landscape.

As a starting point, the FAST Framework will provide the overall structure necessary for lifecycle management, configuration support, and any common and shared modules and tools, shown in Fig. 2 using the problem example from Fig. 1. Discussion of modules, configurations, and containers follows.

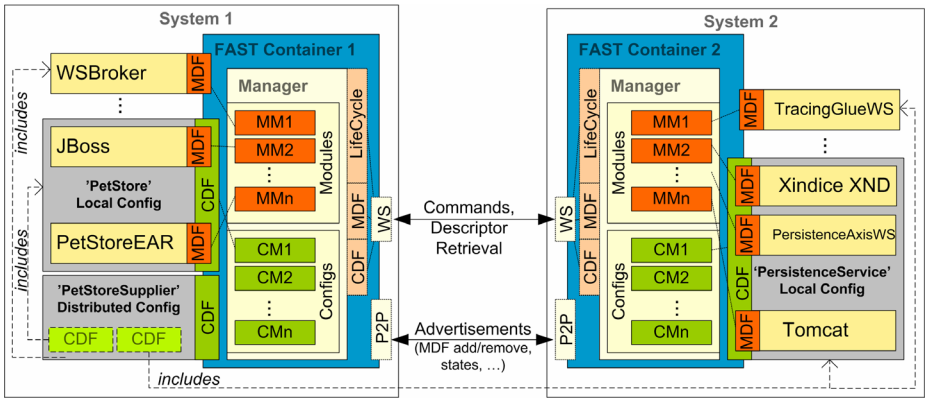

Fig. 2. Sample distributed application with the FAST Framework, showing containers, modules, configurations, and sub-configurations

3.1 Modules

As modularity plays a key role in dealing with complexity, for this paper, a module is an abstraction for the partitioning of a (semi-)independent unit of software (software artifact) required by an application, at any desired abstraction level, and can include frameworks, services, components, packaged applications, web and application servers, documentation, etc. For reuse, extensibility, and flexibility, a module is described via an XML-based Module Descriptor File (MDF), as seen in Fig. 2. This MDF supplies the necessary and desired metadata, typically the information required for its lifecycle management along with any additional information, as shown in the example in Listing 1. Note that an XML Schema definition specifies the allowable values, and is not shown due to space constraints.

Listing 1. Module Descriptor File (MDF) example

```
<module name="PersistenceAxisWS" uid="1422756542">
  <description>
    <category>soa.ws.persistence</category>
    <author>John Doe</author>
    <timestamp> 27-Apr-04 9:12:14.06 GMT</timestamp>
    <version>1.0</version>
...
  </description>
  <dependencies>
    <query id="1" type="module">
      <name>Xindice</name>
    </query>
    <query id="2" type="module">
      <name>Tomcat</name>
      <version min="4.1" max="5.0.19"/>
    </query>
    <query id="3" type="module">
      <name>Axis</name>
    </query>
  </dependencies>
  <management>
    <instances min="0" max="1">
      <instance nbr="1">
        <port>9876</port>
      </instance>
    </instances>
    <lifecycle>
...
      <task type="uninstall">
        <cmd type="ant">
          undeploy/tomcat_undeploywar.xml</cmd>
      </task>
    </lifecycle>
    <templates>
      <template name="axis_webservice" version="1.0">
        <settings>
          <setting name="ws_uri" value="http://${env.ip}:
            ${instance.port}/axis/PersistenceAxisWS"/>
        </settings>
      </template>
    </templates>
  </management>
  <tools>
    <query id="3" type="module">
      <name>Axis TCP Monitor</name>
    </query>
  </tools>
  <documentation>
    <query id="4" type="module">
      <name>Command-Line</name>
      <parameters>
        <parameter>-DCMD</parameter>
        <parameter>docs/PWS-UserGuide.pdf</parameter>
      </parameters>
    </query>
  </documentation>
</module>
```

Each module is given a `name` and a `uid` attribute that allows the unique identification of an MDF. Various options can be used to generate the `uid`, such as tools or the use of a central uid provider. Digital signatures could be included to permit creator verification and detect content tampering. Under `description`, information about the MDF itself is included. The element `dependencies` specifies modules necessary for this module to function properly. Hereby `dependencies` use `query`, which allow constraints to be specified (name, version, location, etc.) and are resolved at runtime to find a matching candidate, e.g., the best, a range, or exactly one required candidate of the available modules known to the Container.

Under `management`, the number of `instances` of this module that can be started as well as any instance-specific parameters can be specified. The module lifecycle transitions can be associated with actions to be performed, and are specified under `lifecycle` with `task` (see Fig. 3). Because all module types should be supportable and creating multiple processes is not always desirable for resource-constrained contexts, different external process (e.g. Apache Ant) and internal process (e.g., same Java Virtual Machine) `cmd` (command) `types`, including parameters, are supported for lifecycle management. One feature (not shown) is the ability to specify a pattern for lifecycle progress and error checking in the output files of external modules.

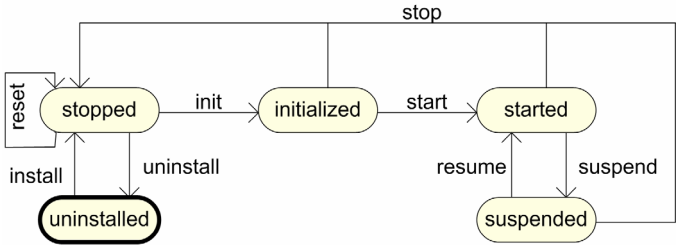

Fig. 3. Module and configuration typical lifecycle state diagram (additional states supported)

A `template`, defined in an XML Schema definition (not shown), allows the inclusion of pre-specified XML, and is a contract in the sense that a module that incorporates `template` indicates that it fulfills its requirements, e.g. requirements for interfaces or protocols that must be supported, pre-configuration of parameters, valid parameter ranges, etc. Templates are analogous in some ways to a class; they can be instantiated with parameters and support inheritance, thus hierarchies of templates are possible as well as overriding of default settings. The `template` contract in a module can be validated against the template's XML Schema definition. In this case, the `axis_webservice` template schema (not shown) includes the AXIS template schema (not shown) which specifies the HTTP, SOAP, and WSDL versions that are supported, and specifies the `ws_uri` value. Note that `setting` is used to set parameters for a software artifact; for artifact configuration settings in text files, scripts (e.g. sed, python) are supported, for XML files XSLT, and JMX is used for runtime (re-)configuration support. While templates are optional, for maximum reuse effectiveness, they should be specified and propagated, e.g. via a central repository. This specification could be done in both general and domain-specific areas via standards bodies, consisting of software vendors, research institutes, etc.

Under `tools`, any tools associated with this module can be included, while `documentation` provides the queries (e.g. commands) or links to retrieve the documentation associated with this module.

MDFs allow the information for a module to be described and associated once, and then reused in many different configurations.

3.2 Configurations

Configurations are a hierarchical composition of modules or sub-configurations and described in a Configuration Descriptor File (CDF), as seen in Fig. 2. A set of queries is used to allow maximum flexibility in specifying and resolving the actual modules or configurations, while minimizing redundant information. Sub-configurations allow a hierarchical reuse in the specification of configurations, e.g. in Fig. 2 both the PersistenceService and the PetStore are sub-configurations of the PetstoreSupplier configuration.

Listing 2. Configuration Descriptor File (CDF) example

```
<configuration name="PetStoreSupplier" uid="4968365039">
  <description>
    <category>application.demo</category>
    <author>John Doe</author>
    <timestamp> 27-Apr-04 9:51:14.06 GMT</timestamp>
    <version>1.0</version>
...
  </description>
  <dependencies>
    <query id="1" type="module">
      <name>TracingGlueWS</name>
    </query>
    <query id="2" type="module" startuporder="1"
        location="localhost">
      <name>WSBroker</name>
    </query>
    <query id="3" type="config" startuporder="1, 4"
        location="strategy_best">
      <name>PetStore</name>
    </query>
    <query id="4" type="config" startuporder="1"
        location="192.168.3.100">
      <name>PersistenceService</name>
    </query>
  </dependencies>
  <management>
    <instances min="0" max="1"/>
    <lifecycle/>
  </management>
  <tools/>
  <documentation/>
</configuration>
```

An example of a CDF is shown in Listing 2. The `name` and `uid` attribute are equivalent to that described for MDFs, as are the `description`, `dependencies`,

`management`, etc. The number of instances allowed of this configuration among a set of containers may be specified by `instances` if it does not conflict with the sum of the underlying MDFs and CDFs constraints.

As to startup ordering, by default parallel or independent lifecycles are assumed unless the `startuporder` attribute is included as an attribute specifying a sequential list of one or more queries that must first be successfully started. Different strategies for the distribution of a configuration can be used. The `location` attribute in a query is optional, and allows either an IP address to be specified, the name of a strategy, e.g. `strategy_best`, or a DNS hostname. If no location is specified, then the Container will decide based on its default strategy. For distributed configurations with unspecified location attributes, the master or hosting Container (the first to start the "init" transition) annotates the location information before distributing the configuration, thus ensuring that other Containers can determine their responsibilities.

The lifecycle of configurations are equivalent to those of modules (see Fig. 3).

3.3 Containers

A Container non-intrusively manages the lifecycle its software modules or configurations. As a form of bootstrapping, its CDF specifies its own core modules. It also supplies its own MDF, as shown in Fig. 2, to describe its capabilities and states. Any extensions to the Container are done as modules via MDFs, providing a plug-in capability. At a minimum (equivalent to an empty Container CDF), the Container contains the software for parsing MDF/CDFs and lifecycle management, allowing it to be lightweight for resource-constrained contexts. Containers can optionally interact via the Container Discovery and related modules to support the distribution of metadata and the discovery of remote configurations and modules. Any inter-Container interaction is done via XML-based protocols to better support interoperability with heterogeneous Container implementations.

4 Solution Realization

While the reference implementation is Java-based, the solution approach can be implemented for a wide variety of platforms and languages while supporting interoperability, due to the reliance on XML for metadata (MDFs and CDFs) and inter-Container protocols (SOAP and JXTA [15]). The following provides insight into the realization while elaborating the potential of this approach. The inclusion of various modules listed below supports basic to enhanced framework configurations depending on the requirements.

4.1 Container

For each module that the Container manages, a Module Manager is allocated, shown as MMn in Fig. 2. Likewise, for each configuration being managed, a Configuration

Manager (CCn in Fig. 2) is allocated. DOM4J was used to parse the needed MDF and CDF information into Java objects. Dependencies are resolved to determine lifecycle sequencing. Support for Apache Ant was integrated as a task type. A Container shell allows command control of the container.

4.2 Modules

Over 50 module descriptions for various software artifacts were created in the current FAST distribution, verifying that it can support many different types of modules. Below are some examples of infrastructural modules, which can be viewed as a Container extensibility mechanism:

- **Container Discovery Module.** Responsible for advertising the existence or change of its modules and configurations to other Containers and detecting other Containers and their state changes. Currently JXTA advertisements are used, however other discovery mechanisms, including registries, can be supported.
- **Container Management Web Service Module.** This optional module supports remote management via SOAP, providing module and configuration descriptor retrieval.
- **Deployer Service Module.** This module supports the deployment of software as a container-independent, distributed, transactional, discoverable deployment service with adapters for various containers (JBoss, OSGi Oscar, Java VM, etc.). A GUI, as shown in Fig. 4, provides operators insight into the location and dependencies within deployment units.
- **Web Services Broker (WSBroker) Module.** An optional module that contains an HTTP Proxy combined with an interception framework that includes the Apache Bean Scripting Framework, allowing Java- or script-based interceptors that provide a variation point for technically trained operators to perform routing, logging, or other functions.
- **Web Services Registry (WSRegistry) Module.** This optional module contains a UDDI-protocol-compliant mechanism to access Web Services.
- **Tracing Web Service (TWS) Module.** This optional module enables the monitoring of operational interactions between modules via built-in or interception (e.g., WSBroker) mechanisms, and made available to tooling for the operational view.
- **Persistence Web Service Module**. This optional module provides a generic Web Service interface to persist data in different XML storage mechanisms.

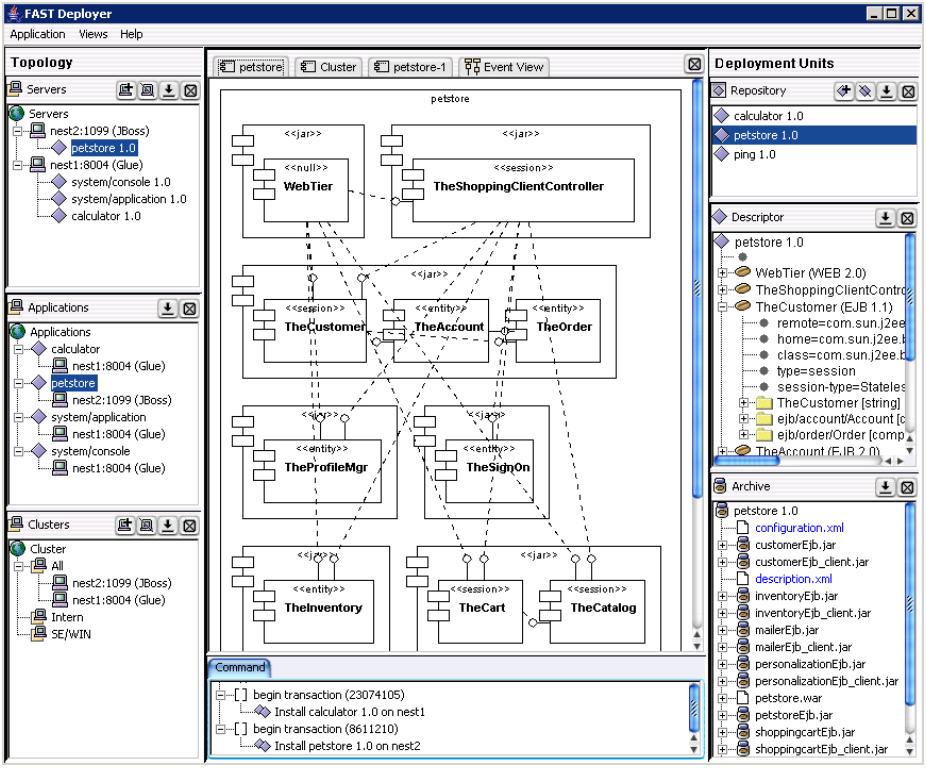

Fig. 4. Screenshot of the FAST Deployer GUI

Solution Results

While many possible criteria could be used to evaluate the solution, tests were chosen that would answer the following questions regarding practical suitability: Does distributed provisioning show significant performance advantages over local provisioning? How does the amount of time for software (re-)deployment (transfer) compare to any potential gain via distribution? How fast does the infrastructure react to faults? What is the memory footprint and scalability profile? How usable was the solution in practice?

The hardware consisted of two Fujitsu Siemens Scenic W600, i8656 (model MTR-D1567) PCs with dual 3GHz CPUs connected by a 100MBit Ethernet LAN and a hub. PC100 had 512MB and PC101 768MB RAM. The software configuration was Windows XP SP1, JXTA 2.2, WebMethods Glue 4.1.2, and Java JDK 1.4.2. Note that no performance or memory tuning was done to the implementation, and for resource-constrained scenarios another XML-based discovery mechanism could be used.

Table 1. Distributed Application Provisioning Test module startup times in milliseconds for a diverse software configuration locally and distributed across 2 PCs.

Module name (time in msec)	Local PC100	Local PC101	Distributed (PC100 except *=on PC101)
Demo Supplier Config. (start)	0	0	0
Jini 1.2.1 HttpServer	1234	2750	1594
Jini 1.2.1 RMIDaemon	1359	4391	1782
Jini 1.2.1 LookupServer	1390	5657	2063
Jini 1.2.1 TransactionManager	1468	6000	2328
James 2.1 Email Server*	4015	4641	5360
TracingWebService	6515	7391	5907
Jini 1.2.1 JavaSpaces	6687	7438	6282
ScriptService	12515	11047	11360
Apache Xindice XML DB 1.0	13234	8250	8750
SupplierOrderWebService	11734	12344	11938
WebServiceBroker*	6671	5797	5516
Jakarta Tomcat 5.0.16	25359	23625	18719
JBoss 3.2.1*	41156	35782	18938
Configuration ready	41656	35782	19375

Table 2. Maximum heap space used by the Container under various conditions

Container Condition (WS=ContainerWebService)	Max Heap Used
A: No modules loaded; WS and JXTA disabled	1MB
B: 50 modules loaded; WS and JXTA disabled	1.2MB
C: No modules loaded; WS enabled; JXTA disabled	2.2MB
D: No modules loaded; WS and JXTA enabled	7MB
E: No modules loaded; WS and JXTA enabled; Peer advertisement of 50 module states received from 2nd Container	7.5MB

The Distributed Application Provisioning Test (Table 1) used 13 modules in a Configuration as shown in Table 1, first measuring their local startup times and then the time when the configuration was distributed across both machines, showing nearly a factor 2 improvement for the application to become ready. By making the distribution of module locations easy, performance gains for startup and shutdown can be reached (due to parallelism). This could improve development cycle efficiencies and application testing realism.

The Deployment Unit Transfer Test determines the amount of time needed to transfer a file by the FAST Deployer into a remote EJB container. The time to remotely deploy from PC100 to PC101 a new 1236KB petstore.ear (modified to run on JBoss) was measured to be 3.4 seconds. While this shows a need for improvement, it does not invalidate the case for distributed provisioning when compared to the shown performance gains for typical software artifacts.

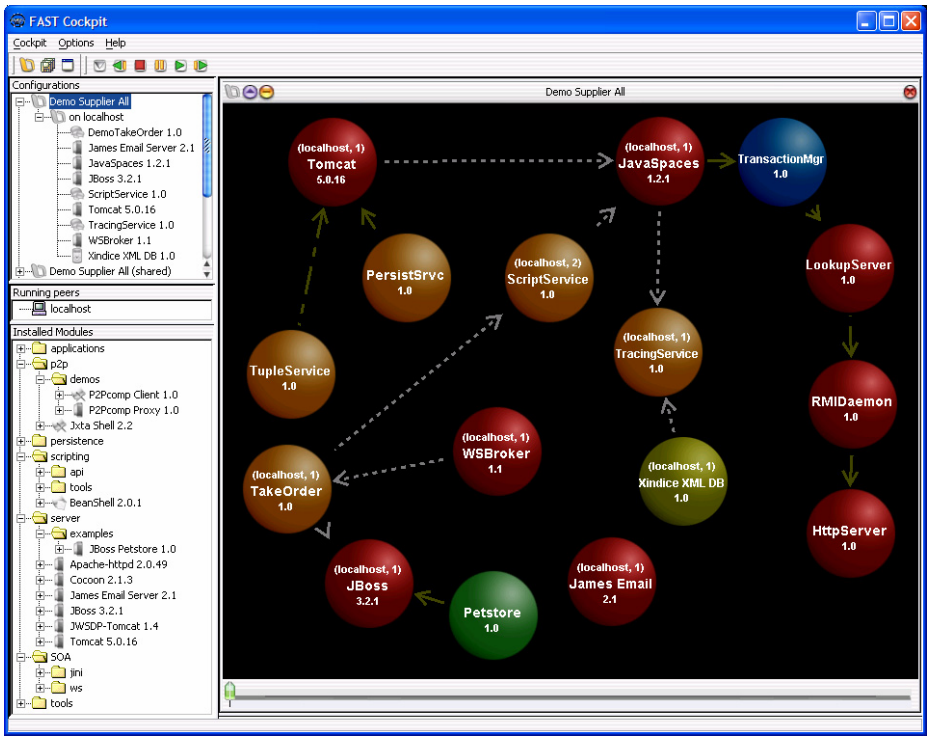

Fig. 5. The FAST Cockpit showing a graphical representation of a configuration of modules (as spheres) with directed lines of interactions and dependencies

The Failure Reconfiguration Test shows the reaction time of the infrastructure to a module failure in a distributed configuration. The James Email Server process was killed on PC100. The reaction time was 200ms from detection of a state change on PC100 through sending of a state-changed advertisement, to receipt on PC101 to the point where it begins to start its local James Email Server module. Thus the infrastructure reaction time would not typically be the primary factor, but rather module startup time.

In the Memory Footprint Test (Table 2), the heap space of the Container on PC100 was measured under various circumstances as shown in Table 2. The difference between condition B and A shows that under most expected scenarios, managing even a large number of modules does not affect the memory footprint significantly. Thus the solution could potentially be applied to resource-constrained contexts.

Other criteria include the experience with usage of the solution realization within the organization. A GUI tool, the FAST Cockpit (see Fig. 5), was developed to demonstrate the operator-friendly possibilities once MDFs, CDFs, and Containers are available. Configurations with their associated modules are listed in the top-left box and can be created by drag-and-drop of installed modules - categorized both in the bottom-left and with sphere colors. Animated tracing with playback is operated with buttons on the top menu and the event slider at the bottom of the configuration. The

status of modules is represented with colored fonts and icons and module outputs are available in separate windows.

An internal distribution among colleagues in the organization has enabled the operation of complex configurations of distributed applications without the operator necessarily being aware of the infrastructural issues and dependencies involved. Based on feedback, a significant improvement in the time required to specify, compose, and instantiate distributed applications has been observed as well as comprehension benefits. Note that the amount of time needed to create an MDF depends on the person's skills and the familiarity with the software being described and MDF concepts, but times of less than 15 minutes have been measured. The time to create a CDF using the Cockpit is a matter of drag-and-drop of modules and any startup sequence dependency specification. Thus, the investment in the one-time MDF creation typically pays off quickly, analogous to the investment in a makefile.

The feasibility, suitability, and advantages of this approach were hereby validated, and future work will continue to improve these results.

6 Conclusion

Despite the growing marketplace competitiveness and pressure for faster software delivery schedules, the challenges with regard to composing, configuring, deploying, and operating distributed applications in a diverse software landscape have not received adequate attention. There remains no unifying, widely adopted, practical, and economic solution.

While the current FAST realization has shown good results, ongoing and future work includes determining to what degree the security, policies, QoS and Service-Level-Agreements can be addressed without annulling the current simplicity and interoperability; correctness; concurrent configuration conflict checking; addressing any single-points-of-failure; performance and memory tuning; a repository for MDF and CDF propagation; a wizard for MDF creation; evaluating the issue of semantics in MDFs and CDFs; and efforts towards more prevalent, standard, or unified software artifact metadata.

FAST presents a practical solution to the current gap with regard to both the goal of distributed application composition in this diverse landscape and given constraints such as effort, cost, and others. A comprehensive solution could be realized by universal software standardization efforts for flexible-granularity and flexible-aspect (e.g. operational) software metadata, in combination with platform, development environment, and tool vendor support for utilizing this metadata.

7 Acknowledgements

The author would like to thank the following individuals for their efforts with regard to various aspects of FAST: Ulrich Dinger, Emmanuel Gabbud, Christoph Huter, Klaus Jank, Josef Pichler, Christian Reichel, and Martin Saler.

References

1. Anderson, P., Beckett, G., Kavoussanakis, K., Mecheneau, G., Toft, P.: Experiences and Challenges of Large-Scale System Configuration. (2003) http://www.epcc.ed.ac.uk/gridweaver/
2. Oberleitner, J., Gschwind, T., Jazayeri, M.: The Vienna Component Framework: Enabling Composition Across Component Models. Proceedings of the 25th International Conference on Software Engineering (ICSE). IEEE Press (2003)
3. Burmester, S., Giese, H., Niere, J., Tichy, M., Wadsack, J., Wagner, R., Wendehals, L., Zuendorf, A.: Tool Integration at the Meta-Model Level within the FUJABA Tool Suite. In Proc. of the Workshop on Tool-Integration in System Development (TIS), ESEC/FSE 2003 Workshop 3. Helsinki, Finland (2003)
4. Lüer, C., van der Hoek, A.: Composition Environments for Deployable Software Components. Technical Report 02-18. Department of Information and Computer Science, University of California, Irvine (2002)
5. Goldsack, P., Guijarro, J., Lain, A., Mecheneau, G., Murray, P., Toft, P.: SmartFrog: Configuration and Automatic Ignition of Distributed Applications. HP OVUA (2003)
6. Jini Rio: http://rio.jini.org/
7. Object Management Group: Model-Driven Architecture (MDA) - A Technical Perspective, ormsc/2001-07-01 edition (2001)
8. Gokhale, A., Schmidt, D., Natarajan, B., Wang, N.: Applying Model-Integrated Computing to Component Middleware and Enterprise Applications. Special issue of Communications of ACM on Enterprise Components, Services, and Business Rules, Vol 45, No 10, Oct (2002)
9. Wang, N., Natarajan, B., Schmidt, D., Gokhale, A.: Using Model-Integrated Computing to Compose Web Services for Distributed Real-time and Embedded Applications. www.cs.wustl.edu/~schmidt/PDF/webservices.pdf
10. Gokhale, A., Natarjan, B., Schmidt, D., Wang, N., Neema, S., Bapty, T., Parsons, J., Gray, J., Nechypurenko, A.: CoSMIC: An MDA Generative Tool for Distributed Real-time and Embdedded Component Middleware and Applications. In Proceedings of the OOPSLA 2002 Workshop on Generative Techniques in the Context of Model Driven Architecture. ACM, Nov. (2002)
11. Kon, F., Roman, M., Liu, P., Mao, J., Yamane, T., Magalhaes, L., Campbell, R.: Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In Proceedings of the Middleware 2000 Conference. ACM/IFIP, Apr. (2000)
12. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J.: Aspect-Oriented Programming. In Proceedings of the 11th European Conference on Object-Oriented Programming, June (1997)
13. Foster, I., Kesselman, C., Tuecke, S.: The Anatomy of the Grid: Enabling Scalable Virtual Organizations. International Journal of Supercomputer Applications and High Performance Computing, 15(3) (2001)
14. WS-Resource Framework (WSRF) http://www.globus.org/wsrf/
15. JXTA: http://www.jxta.org