Reducing Enterprise Java Bean Deployment Costs via Model-Driven Deployment and Configuration

Jules White and Douglas C. Schmidt,

Vanderbilt University, Department of Electrical Engineering and Computer Science,
Box 1679 Station B, Nashville, TN, 37235
{jules, schmidt}@dre.vanderbilt.edu
http://www.dre.vanderbilt.edu

Abstract. *Extensive testing is required to develop reliable distributed Enterprise Java Bean (EJB) applications. It is therefore crucial to create test environments that ensure the quality of these applications on multiple OS platforms and hardware configurations. Creating separate test environments for different developers and/or development teams makes it easier to rapidly refactor components and test them in a known working system configuration without interrupting other test configurations. Once an application is deemed ready for deployment and configuration in a production environment, it is crucial that these activities be done identically to the tested configurations and upholds the assumptions of the component developers. Rapidly setting up numerous distributed test environments and ensuring that they are deployed and configured correctly is hard. In this paper we present Ant Hill, which is a tool for the model driven development of deployment plans, and Fire Ant, which is a tool for remotely deploying distributed EJB applications. A distributed constraints optimization system for scheduling highway freight shipments to trucks is used as a case-study to illustrate the significant improvements in deployment correctness, reproducibility, and manual efforts gained from the use of these tools.*

1 Introduction

Component-based distributed systems require extensive testing to ensure proper functionality. Testing these types of systems involves deploying and configuring the application components across a group of nodes connected via a network. In many cases, deploying and configuring application components requires physical access each node to install the necessary software and configure it properly. Since testing can account for as much as 2/3 of application development cost [1], it is crucial that the deployment and configuration of test environments be straightforward and effective .

Much deployment and configuration of application components today is done with *ad hoc* techniques, such as having a system administrator remotely copy files to a group of computers, editing XML configuration files, and starting an application server. Moreover, deploying a distributed application component to a node often requires significant insight into the configuration. Since distributed applications may consist of a hundred or more components, these ad hoc techniques make it hard to manage the large number of steps and artifacts required to deploy and configure a component-based distributed application.

Ad hoc techniques often employ build and configuration tools, such as Make and Another Neat Tool [2], but application deployers still must manage the large number

of scripts required to perform the component installations. Developing these scripts can involve significant effort and require the in-depth understanding of the components. Understanding the intricacies and properly configuring the application is crucial to its proper functionality and quality of service (QoS) requirements [3]. For example, the performance of an Enterprise Java Bean (EJB) [4] component can be greatly affected by the object pool settings of an application server. Setting the pool's minimum size too low can lead to failed requests under heavy loads, whereas a minimum size that is set too high wastes application resources and may adversely affect other components. Incorrect system configuration due to operator error has been shown to be a significant contributor to down-time and recovery [5].

Developing custom deployment and configuration scripts for each application leads to a significant amount of reinvention and rediscovery of common deployment and configuration processes. The scripts themselves can become sources of application error. Many component installations may require significantly different deployment and configuration processes for each target environment. A script that fails to account for each of the intended target and component configurations can become a source of application errors.

Deployment and configuration scripts require human interaction and thus there is no assurance of the quality of an installation. Often, errors in the configuration process can create hard to debug errors and require costly expert attention to correct. Operator and deployer errors are some of the most significant sources of distributed application down time. Requiring operator interaction also degrades the repeatability of the deployment and configuration process since there is no guarantee that an operator correctly runs the deployment and configuration process for each installation. *Ad hoc* deployment and configuration methods needing human intervention, provide no quality assurance and make it hard to reproduce deployments predictably.

This paper presents three contributions to the deployment and configuration of distributed EJB applications. First, we describe the structure and functionality of *Ant Hill*, which is an open-source model-driven Eclipse plug-in for specifying and generating deployment and configuration plans for EJB applications. Second, we describe the structure and functionality of *Fire Ant*, which is an open-source tool for executing deployment plans generated by Ant Hill. Third, we illustrate the improvements in deployment script creation, deployment correctness, and deployment reproducibility provided by using these two tools in the context of a case study of an EJB-based system that schedules highway freight shipments using the multi-layered architecture shown in Figure 1. The system has a list of freight shipments that it must schedule using a constraint-optimization engine to find a cost effective assignment of drivers and trucks to shipments.

The highway freight shipment system is composed of the *Scheduler Module, Request Module, and Route Time Module*. Each of these modules contains beans that must be deployed and configured on separate application servers. Each module also has specific requirements for the application server on which it is deployed. For example, the *Request Module* contains beans that require access to the database containing shipment requests.

A central component in Figure 1 is the *Route Time Module* (*RTM*), which determines the route time from a truck's current location to a shipment start or end point.

The *RTM* uses a geo-database and the GPS coordinates from the truck to perform the calculation. This module is critical to the proper operation of the optimization engine. A heavy load is placed on the *RTM*, so it is crucial to configure it properly. Another key aspect of the *RTM* is that it contains two beans, the *RTM Bean* and *Truck Locations Bean*, that must be co-located on the same application server to provide adequate performance. Moreover, the *Truck Locations Bean* must reside on an application server with fast access to the geo-database containing the current truck locations.

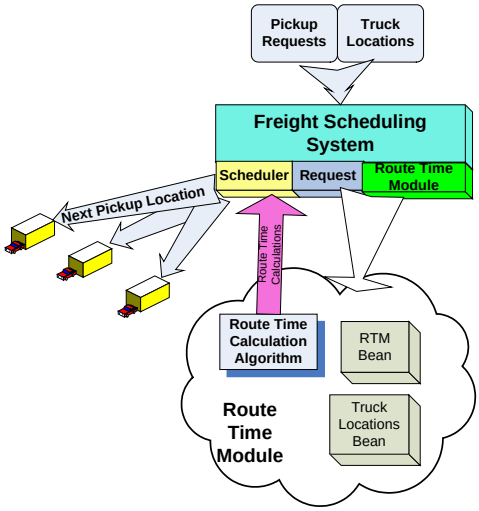

Figure 1: Highway Freight Shipment Scheduling Architecture

The remainder of this paper is organized as follows: Section 2 describes the Ant Hill model-driven tool for EJB component deployment; Section 3 describes the Fire Ant Deployer; Section 4 quantifies the reduction in manual deployment effort achieved by applying Ant Hill and Fire Ant to our highway freight shipment case study; Section 5 compares our work with related research; and Section 6 presents concluding remarks.

2 Improving Deployment Plan Quality via Ant Hill

The effectiveness of deployment and configuration capabilities can be measured by three metrics: (1) the ***correctness*** of the deployment plan, i.e., whether or not the configuration and deployment of each component is accounted properly for, (2) the *execution accuracy* of the system, i.e., whether or not the plan is executed properly, and (3) the *execution reproducibility*, i.e., whether or not the plan can be executed repeatedly with the same results.

To address the challenges of deploying and configuring EJB applications, we have created *Ant Hill* and *Fire Ant*. Ant Hill is a model-driven tool designed to improve deployment plan correctness by (1) visually capturing the components that need to be deployed and their dependencies, (2) describing the target infrastructure required for the deployment, (3) visually specifying which components are to be deployed and configured on which machines, (4) visually specifying the configuration of the com-

ponents on each target, and (5) providing constraint checking to ensure that each component and its dependencies are accounted for properly. Fire Ant takes the deployment models produced by Ant Hill does the run-time execution of the deployment plan to provide execution accuracy and reproducibility.

2.1 Ant Hill

Ant Hill is a model-driven tool designed to allow developers to create deployment and configuration plans for EJB systems that are correct by construction. Ant Hill provides a domain-specific modeling language (DSML) that allows developers to visually specify the EJBs required for a distributed application, their configuration, their application server configuration, and their target nodes. Another capability of Ant Hill that is crucial to creating correct deployment plans is its constraint-checking to ensure that plans are constructed properly.

Ant Hill is designed to facilitate the role of a *deployment planner*, who determines the allocation of components to physical application nodes. A deployment planner also determines the software installation and configuration process that must be performed to deploy each component on its target node. This basis installation and configuration includes setting up an application server (such as JBoss) on the target environment and editing its configuration files to establish needed database connections.

Ant Hill was developed using the Generic Eclipse Modeling System (GEMS) [6] created by the Distributed Object Computing (DOC) Group at the Institute for Software Integrated Systems (ISIS) at Vanderbilt University. GEMS is an model-driven environment built using Eclipse. A GEMS-based metamodel [7][8] describing the problem domain was constructed and interpreted to create the Ant Hill DSML for deployment and configuration of EJB systems. Ant Hill models constructed in the domain are interpreted to produce Fire Ant build scripts. This approach is similar to other model-driven efforts that the DOC group has used for the deployment and configuration of component systems in prior work [9, 10, 13, 15].

2.2 Ant Hill Application Model

To develop an Ant Hill deployment plan, a deployment planner must first create an *application model*, which describes the EJBs and their required physical artifacts that Fire Ant will deploy. Creating the *application model* includes capturing the EJBs, their deployment descriptors, and other resources (such as supporting jar files) required by the EJBs. Deployment planners can create this model manually, by dragging and dropping components in the Ant Hill modeling environment, or by importing a jar file containing compiled Java class files, EJB deployment descriptors, and *Fire Ant Requirement descriptorS* (FARS). FARS contain constraints associated with each bean that must be fulfilled in the environment where the bean is deployed. For example, a message bean may declare in a FARS that it requires access to a specific message queue. FARS allow component developers to pass detailed configuration requirements to the deployment planner.

Creating an application model provides Ant Hill with a list of resources that it must configure and deploy properly. This list allows deployment planners to eliminate common sources of error in deployment plans, such as failing to configure a

component or deploy a component properly. In an Ant Hill application model, every component must have an associated configuration, which eliminates errors arising from failing to configure resources. All components must also be deployed to a target node, eliminating errors stemming from undeployed components.

The Ant Hill application model consists of packages of components, called *assemblies*, that need to be deployed to different application servers. Each assembly contains one or more EJBs, EJB deployment descriptors, and supporting EJB resources. The models of the EJBs contain the constraint information imported from the FARS, such as required database connections and message queues. Figure 2 shows the portion of the Ant Hill application model for the constraints optimization system, discussed in Section 1, that contains the Route Time Module's EJBs.

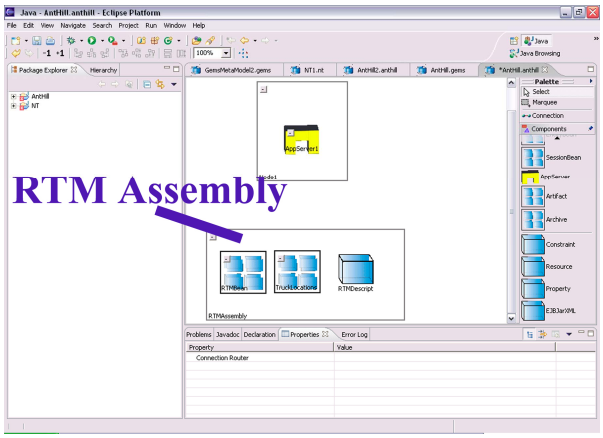

Figure 2: RTM Assembly

The Route Time Module has one assembly, containing the RTMBean EJB, TruckLocations EJB, and the RTMDescriptor EJB deployment descriptor. Figure 3 illustrates the *TruckLocations* EJB's constraint that it must have JDBC access to the truck location database.

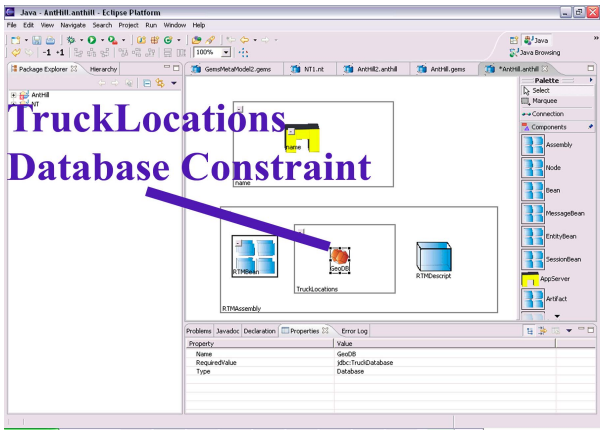

Figure 3: TruckLocations EJB Database Constraint

Assemblies ensure that components and their dependencies are deployed together. By grouping components into assemblies, Ant Hill ensures that components are not deployed to a target without their required dependencies, such as a supporting library. Assemblies also ensure that components that require collocation are deployed to the same target node. When dozens or hundreds of components are being deployed, Ant Hill assemblies significantly reduce the complexity of tracking and managing dependencies. Assemblies also help improve plan correctness by eliminating frequent and possibly hard to diagnose errors in component deployment plans.

Importing Java class files is the preferred method of creating the application model since it ensures that the EJB application is properly represented in the Ant Hill model, thereby enhancing plan correctness. The Jar importation tool uses the Java `instanceof` operator to identify classes that implement the `SessionBean`, `EntityBean`, and `MessageDrivenBean` interfaces. Each class that implements one of these interfaces is added as the corresponding type in the Ant Hill model. The importation tool also identifies the name of the Jar file containing the compiled class file for each bean in the application model.

Deployment planners can easily introduce typographic errors in the naming of EJBs or other artifact names that would not be detectable until the deployment plan was actually executed. Importing compiled Java code saves deployment planners the significant effort that would otherwise be required to model a large-scale EJB distributed application in Ant Hill. Importation also allows development groups to separate the roles of component development and deployment planning, since the deployment planner only needs to access an archive containing (1) compiled classes, (2) the FARS describing the EJB deployment constraints, and (3) a list of the EJBs that must be deployed together as assemblies.

Even with the many advantages of using class file importation as the basis for creating the application model, there are circumstances in which manual construction of the application model is required. In some cases the compiled class files for the components may not be available to the system deployment planner. This situation could arise if the EJBs were being developed in parallel to the deployment plan or if an existing deployment plan was modified in anticipation of forthcoming functionality.

Once the EJBs for an application have been modeled, the deployment planner may then begin specifying additional resources that should be deployed along with the beans. As discussed later in Section 3, Fire Ant uses a single archive, called an *EGG*, containing all required physical artifacts as input to the Fire Ant deployer. Each additional resource is given a URI, relative to the root of the *EGG*, specifying where the resource resides. At deployment time, Fire Ant uses this URI to locate the resource. In the constraint optimization system, a web application front end is deployed for each system module. This front end includes Java Server Pages and Java Servlets that allow system administrators to gather critical system information on each module, such as the number of pending route time requests currently queued by the RTM.

The *EGG* single archive enhances both plan correctness and execution correctness by allowing the runtime deployment infrastructure to read the deployment plan and check that all the required artifacts are present. This verification process allows the Ant Hill runtime deployment system to detect missing artifacts before (1) a build is executed and (2) possibly hard to reverse changes are applied to the target node.

After the EJBs and related resources have been modeled, deployment planners must assemble the artifacts needed to construct the Fire Ant *EGG*. Ant Hill provides a model interpreter that can create the expected directory structure for the *EGG* based on the information specified in the model. This interpreter creates the directories for EJB jar files, EJB deployment descriptors, and additional resources files. Within each of the generated directories, a file containing the list of expected artifacts is also created. Generating the required directory structure and file lists helps enhance plan correctness by ensuring that deployment planners construct the *EGG* properly.

2.3 Ant Hill Configuration and Deployment Model

After constructing the application model, deployment planners must specify how each assembly is configured and where to deploy it. Configuration and deployment is captured in the *Deployment and Configuration model (*DnC model*)*. This model allows deployment planners to specify the physical node where each assembly will reside and the extra configuration that must be done to the physical node for the assembly to function properly.

The first step in creating the Ant Hill DnC model is to develop a model of different physical resources required for the deployment. Deployment planners drag and drop *nodes* into the Ant Hill model for each physical node required. For each node added to the model, developers must specify its *provided properties*, which are resources that a node makes available to an assembly. For nodes, provided properties could include CPU speed, CPU count, or available RAM. After creating the nodes, a deployment planner drags and drops *application servers* into the nodes. Application servers correspond to EJB application servers that will be running on the physical node. Generally, each node will contain exactly one application server, but Ant Hill allows multiple application servers per-node to provide flexibility. Each application server must have its provided properties modeled, such as its available database connections.

The node and application server specifications are not tied to specific machines or application servers. The models serve as a requirements list for the actual physical machines and application servers that the application will be deployed to. These re-

quirements are for planning purposes and can be used to verify the correctness of the target environments at deployment time. At deployment time, the nodes are mapped to actual physical nodes, which improves execution accuracy since the runtime deployment system can ensure that the deployment plan properly accounts for the target environment.

To associate assemblies with application servers, the deployment planner creates connections between the two in the DnC model. Ant Hill matches the constraints contained within the assemblies against the provided properties of the application server and node to ensure that the resources required for the proper functioning of that application component are met. If the application server and its hosting node do not contain provided properties matching the constraints, Ant Hill prevents the connection from being made. Matching constraints in this manner ensures the nodes that components in assemblies are deployed to meet their resource requirements and is vital to the application's proper functionality [10][11]. Figure 4 shows the geo-database constraint of the *TruckLocations* EJB being matched to the geo-database provided property of an application server. This feature helps Ant Hill improve both plan correctness and execution accuracy since only plans that deploy components to nodes satisfying their infrastructure requirements can be constructed. Moreover, once a plan is constructed and executed, the runtime deployment system can ensure the target infrastructure matches that modeled in the deployment plan.

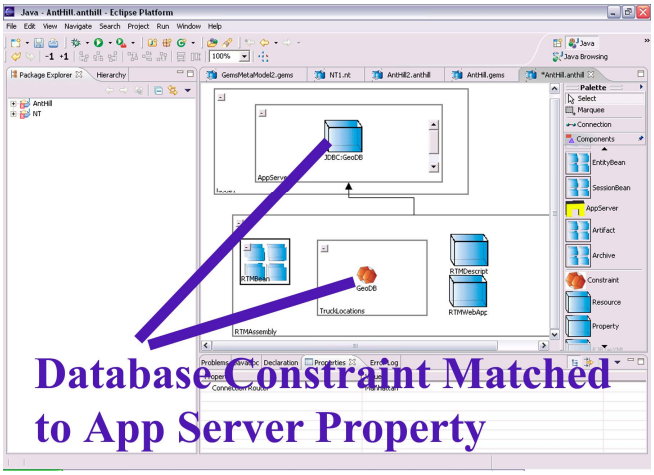

Figure 4: A Database Constraint Matched to a Database Provided Property

EJB applications often require special preparation of the target environment and application server, including the establishment of specific database connections by the application server or the creation of a required set of directories on the target node. To facilitate this custom configuration, Ant Hill provides a mechanism for specifying ANT scripts that must be run on the target environment before and after the deployment of assemblies. Deployment planners drag and drop *ANT resources* into the DnC model to represent these custom configuration scripts. Each Ant resource has a URI specifying its location within the EGG and a URI, relative to the

assembly installation directory, specifying the root directory for the script on the target environment. The script also specifies if it should be run before or after the assembly is deployed.

After the deployment planner has finished constructing the DnC model, a model interpreter is run to generate the Fire Ant deployment script for the model. The script is used by Fire Ant to map the assemblies to the target nodes at deployment time. It is also used to determine the ANT scripts to run on the target environment before and after assembly deployment. Fire Ant scripts are discussed in more detail in the following section. Using a model interpreter to generate the script eliminates typographical and programmatic errors that could be made if the script was developed manually, which is another improvement of plan correctness.

3 Design of the Fire Ant Deployer

Fire Ant is a model-driven tool for deploying a distributed EJB application to one or more target nodes. It is designed to provide execution accuracy and reproducibility. Fire Ant is based on the open-source ANT build tool developed by the Apache Foundation [2] and is designed to ensure that deployments of distributed EJB application components can be done reproducibly to different target nodes with a minimum of effort by eliminating human errors in the execution of an application deployment and configuration.

We chose ANT as the basis for Fire Ant since it provides a broad range of built-in deployment and configuration tasks. It also a widely used tool for Java build, deployment, and configuration. Moreover, ANT's wide use and strong industry support make it stable, which is essential to execution correctness and reproducibility.

Fire Ant uses a package format called an *EGG*, which contains all the artifacts required for a deployment and the Fire Ant deployment plan. The Fire Ant deployment plan is an XML file that orchestrates the deployment of one or more assemblies of EJBs to their target locations. The Fire Ant deployment plan specifies what artifacts in an EGG to deploy on each node. The deployment plan also specifies additional ANT build scripts that run on each target node to perform special pre- and post-assembly deployment configuration.

A Fire Ant deployment proceeds in the following steps shown in Figure 5:

1. Fire Ant is launched.
2. Fire Ant then parses the deployment plan and verifies that all required resources are present in the *EGG*.
3. Fire ANT copies an ANT installation to the target using Secure Copy Protocol (SCP).
4. Fire ANT copies the *EGG* to the target using SCP.
5. Fire ANT executes any required pre-deployment configuration ANT scripts.
6. Fire ANT executes an ANT build, which deploys the components in the Assembly to the application server residing on that node.
7. Fire ANT executes any post-deployment configuration ANT scripts.
8. Fire ANT deletes the ANT installation and *EGG*.
9. Fire ANT closes the SSH connection to the target
10. Steps 4-10 are repeated for each deployment target.

Fire Ant uses SSH for its remote deployment, which provides significant advantages in terms of security, target pre-configuration, and manual intervention. SSH is an established secure standard for communicating with remote systems. Communicating over another protocol would require opening additional ports in an organization's firewall and the implementation of a secure authentication system. The secure authentication system itself would need to be installed on the target node and would increase the target node's public points of malicious attack. SSH also allows Fire Ant a trusted, well tested, and accepted framework for doing remote deployment. Moreover, SSH access is integrated into ANT and does not require extra development work.

The only initial configuration that a target node needs to receive a deployment is a working Java Runtime Environment (JRE). Although it is possible to push one to the client using Fire Ant, this is not an advised practice, so we recommend that a JRE be installed and tested before deploying components. It is also unlikely that a target node for an EJB application would not already have a functioning JRE installation and require one to be pushed through by Fire Ant. Requiring only a functioning JRE alleviates developers from having to correctly deploy and install a custom deployment base. JREs are well understood and much more likely to be properly deployed and configured, which is key to execution correctness and reproducibility, than a custom deployment solution.

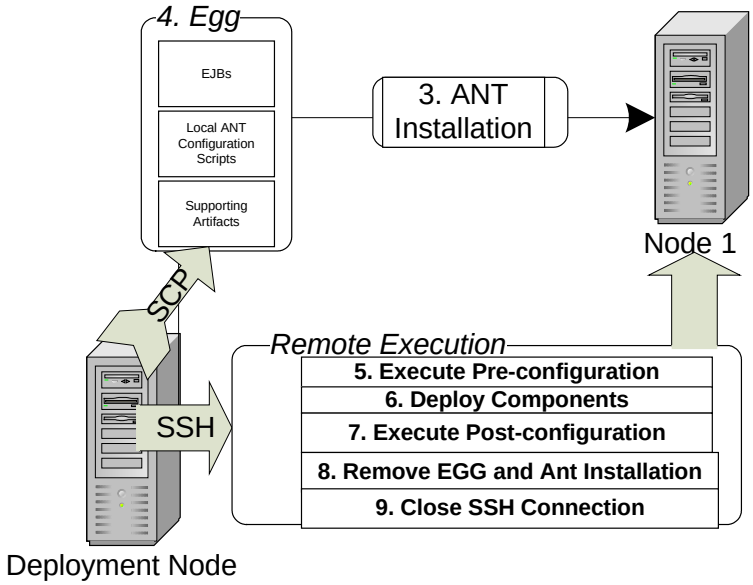

Figure 5: Fire Ant Deployment Process

Although Fire Ant does not provide a built-in means of checking that every possible required resource on a target node is fulfilled, pre- and post-installation ANT scripts can be used for this purpose. A pre-configuration ANT script can be developed to ensure that the target environment meets its requirements. If the pre-configuration script fails, deployment will not proceed.

4 Case Study

Our constraints optimization system for scheduling highway freight shipments shown in Figure 1 required several groups of EJBs to be distributed across multiple applications servers. Each of the EJBs composing the system had requirements that needed to be met in this target environment. We developed three separate deployment and configuration processes to compare (1) a fully manual deployment and configuration of our constraints optimization system, (2) a solution based on ANT, and (3) a solution based on Ant Hill (and Fire Ant). We then compared the number of manual steps involved for each, as well as the number of lines of scripting code that was written for each.

The manual and Ant Hill approaches required writing no scripting code. The Ant Hill approach generated approximately 600 lines of Fire Ant scripting code that required no manual editing. In contrast, the ANT approach required handcrafting an equivalent amount of ANT script code. This difference was expected since Fire Ant is based on ANT, uses the same XML file format, and shares many of ANT's tasks. The 600 lines of generated Fire Ant code correct-by-construction. Since the ANT code was written manually it therefore required debugging.

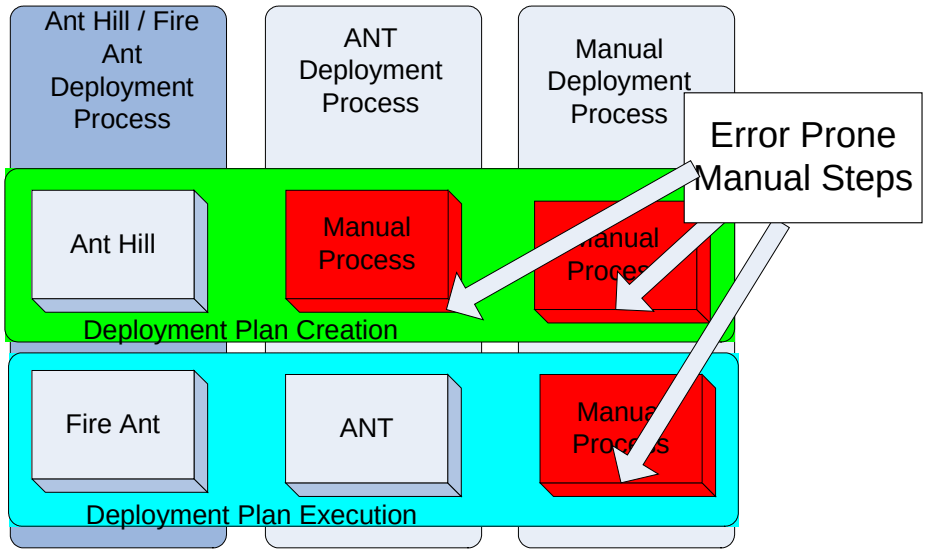

Figure 6: Comparison of Deployment Processes in Terms of Manual Steps

We next evaluated the total number of manual steps needed for each deployment. The manual deployment took approximately 50 manual tasks executed on three sepa-

rate target nodes. The ANT and Fire Ant deployments only required one initial manual task to launch the deployment.

Finally, we examined what types of errors could occur in each deployment process to reduce plan correctness, execution accuracy, and execution repeatability. Figure 6 illustrates the error-prone manual steps involved for each of the three processes. The manual approach yielded the lowest rankings on each metric since human intervention was needed at each step of the process. Moreover, there was no assurance that the deployer would correctly account for the configuration and deployment of each component, correctly match the component requirements to the target nodes, execute the deployment plan correctly, and execute the deployment plan the same way in multiple executions. The ANT approach, however, ensured execution accuracy and execution repeatability, but did not provide any plan correctness assurance. In particular, human development of XML ANT scripts can yield bugs that are hard to find. Moreover, neither the ANT approach nor the manual process did any constraint checking to ensure that the components were deployed and configured properly on suitable infrastructure. Conversely, the Ant Hill approach provided constraint checking to match component requirements, dependencies, and target capabilities.

5 Related Work

Fire Ant is inspired by the Deployment and Configuration specification [12] for the CORBA Component Model (CCM). Both provide the ability to remotely deploy and configure a distributed application to a group of nodes. The two, however, have some significant differences. The OMG Deployment and Configuration specification for CCM focuses on component deployment and assumes that the application server on the target environment is already configured properly. This specification provides mechanisms to adapt the container of the application but not the application server itself. In contrast, Fire Ant provides the ability to run arbitrary pre- and post-installation configuration steps, which gives deployers the ability to install and configure auxiliary applications or even the application server itself as part of the deployment process. This capability can be useful for testing purposes where it is essential that the deployment and configuration process ensure that application servers are configured properly during each test cycle. Fire Ant also does not require that a daemon application be installed on the target environments, but instead can use SSH to copy over all its required infrastructure, which reduces the burden on deployers.

Deployment tools exist that provide the ability to separately model components and the physical nodes they run on. One example is Proactive [18], which is a distributed programming framework for deploying object-oriented grid applications that models applications as virtual structures and removes references to the physical machines from the functional code of the components. The functional code is later mapped to physical machines using XML descriptors. Proactive separates the modeling of components and targets but does not provide the extensive component dependency and overall application correctness checking that Ant Hill supports. Proactive also does not include as flexible of a runtime deployment infrastructure as Fire Ant.

Other modeling tools exist for developing deployment plans. The CoSMIC [13][14] tool suite provides the ability to specify deployment plans for CCM applications. Ant Hill provides similar functionality to CoSMIC, but for EJB rather than

CCM applications. CoSMIC is a tool based on the Generic Modeling Environment (GME) [7] and is only available for Microsoft Windows. In contrast, Ant Hill is based on the Generic Eclipse Modeling System (GEMS), which is based on the platform-independent Eclipse Integrated Development Environment (IDE).

[19] proposes using UML to model the deployment and configuration of components on application servers. This approach, however, lacks the deployment script generation capabilities of Ant Hill. Without this generative capability, there is no assurance that the actual implementation of the deployment and configuration will be done according to the model. Ant Hill also provides a DSML that is specific to the deployment and configuration of components and thus provides greater expressive capabilities than generic UML.

Model-driven component design tools, such as Cadena [15] and J2EEML [16][17], exist for Eclipse. J2EEML is a model-driven development tool for designing EJB systems that provides the capability to package components, generate build scripts for them, and generate test infrastructure. Its packaging infrastructure and generation of ANT build scripts is similar to some of Ant Hill's functionality. J2EEML, however, is designed to package EJBs into EAR archives and does not generate any deployment scripts. It does generate the deployment descriptors for the EJBs. Ant Hill and Fire Ant are designed to start from EAR and Jar files and produce reproducible and correct deployment and configuration processes for their contained components.

It is also worth comparing Fire Ant and Ant Hill to ANT since it is the basis of these two tools. ANT does not provide the complicated constraint checking and deployment correctness checking provided by Fire Ant. It is possible to create this functionality with ANT, as we have done, but it requires significant effort. Moreover, ANT does not provide any model-driven tools to create component deployment plans. Although graphical editors do exist for ANT, they are general tools not strictly designed for component deployment and configuration.

6 Concluding Remarks

Deploying and configuring component-based distributed applications presents significant R&D challenges. With traditional methods of deployment, where human administrators use *ad hoc* techniques to do most of the actual component installation and configuration, there is a significant risk that deployments will be erroneous. For example, misconfiguration can result in subtle application bugs that do not manifest themselves immediately, but can lead to costly system down time and defects.

Using model-driven techniques to create deployment plans and tools, such as Ant Hill and Fire Ant, significantly reduces the probability that deployment and configuration will be done improperly. These tools also allow deployment to be a separate role from component development. Developers need only supply component resource requirements, in a form such as FARS, to the deployment planners. Our Ant Hill modeling tool can then ensure that the planned deployment meets the requirements of the components. Ant Hill also provides other consistency checks to reduce the number of errors in the deployment process.

Building a correct deployment plan only provides part of the solution since there still must be assurance that the plan executes properly. Our Fire Ant modeling tool fills this role by alleviating deployers from manually executing each step in a deploy-

ment plan. Fire Ant also ensures that a deployment plan will always execute in exactly the same manner. A deployment executor improves a development effort's ability to diagnose errors, generate test environments, and deliver an installation solution.

A key aspect of correct deployments is ensuring that the target environment and the expectations of the components are consistent. Fire Ant provides a method of matching these parameters but they still must be entered manually. Moreover, Fire Ant cannot guarantee at runtime that every requirement specified for a target node is met. Currently, the node requirements are a guideline used by administrators. In future work, therefore, we plan to automate the discovery of target node provided resources and the checking of target environments for these resources at deployment time. The Fire Ant deployer and Ant Hill deployment planning tool are open-source projects available from http://www.dre.vanderbilt.edu/~jules/Fire Ant.html.

References

1. Harrold, M. J., Liang, D., Sinha, S.: An Approach to Analyzing and Testing Component-Based Systems. In: Proc. ICSE'99 Workshop on Testing Distributed Component-Based Systems (1999)
2. Apache Foundation: Apache Ant. http://ant.apache.org
3. Wang, N., Gill, C., Schmidt, D., Subramonian, V.: Configuring Real-time Aspects in Component Middleware. In: Proc. of the Conference on Distributed Objects and Applications (DOA 2004), Cyprus, Greece
4. Matena, V., Hapner, M.: Enterprise Java Beans Specification, Version 1.1. Sun Microsystems (1999)
5. Oppenheimer, D., Ganapathi, A., Patterson, D.: Why do Internet Services Fail, and What can be Done about It?, In: Proc. USENIX Symposium on Internet Technologies and Systems (2003)
6. GEMS, The Distributed Object Computing Group, Vanderbilt University, http://www.dre.vanderbilt.edu/~jules/gems.html
7. Ledeczi, A., Bakay, A., Maroti, M., Volgysei, P., Nordstrom, G., Sprinkle, J., Karsai, G.: Composing Domain-Specific Design Environments. IEEE Computer, Nov. (2001)
8. Ledeczi, A.: The Generic Modeling Environment. In: Proc. Workshop on Intelligent Signal Processing (2001), Budapest, Hungary
9. Edwards, G., Deng, G., Schmidt, D., Gokhale, A., Natarajan, B.: Model-driven Configuration and Deployment of Component Middleware Publisher/Subscriber Services. In: Proceedings of the 3rd ACM International Conference on Generative Programming and Component Engineering (2004), Vancouver, CA
10. Krishna, A., Turkay, E., Gokhale, A., Schmidt, D.: Model-Driven Techniques for Evaluating the QoS of Middleware Configurations for DRE Systems. In: Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium (2005), San Francisco, CA
11. Dearle, A., Kirby, G.N.C., McCarthy, A.J.: A Framework for Constraint-based Deployment and Autonomic Management of Distributed Applications. In: Proc. IEEE International Conference on Autonomic Computing (2004), New York, NY

12. Object Management Group: Deployment and Configuration Adopted Submission. OMG Document ptc/03-07-08 edn. (2003)
13. Gokhale, A., Balasubramanian, K., Balasubramanian, J., Krishna, A., Edwards, G., Deng, G., Turkay, E., Parsons, J., Schmidt, D.: Model Driven Middleware: A New Paradigm for Deploying and Provisioning Distributed Real-time and Embedded Applications. Elsevier Journal of Science of Computer Programming: Special Issue on Model Driven Architecture, Edited by Mehmet Aksit, 2005 (to appear)
14. Balasubramanian, K., Balasubramanian, J., Parsons, J., Gokhale, A., Schmidt, D.C.: A Platform-Independent Component Modeling Language for Distributed Real-time and Embedded Systems. In: Proc. of the 11th IEEE Real-Time and Embedded Technology and Applications Sym. (2005), San Francisco, CA
15. Hatcliff, J., Deng, W., Dwyer, M., Jung, G., Prasad, V.: Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems. In: Proceedings of the 25th International Conference on Software Engineering (2003), Portland, OR
16. White, J., Schmidt, D., Gokhale, A.: The J3 Process for Building Autonomic Enterprise Java Bean Systems. In: Proceedings of the International Conference on Autonomic Computing (ICAC 2005), Seattle, WA
17. White, J., Schmidt, D., Gokhale, A.: Simplifying the Development of Autonomic Enterprise Java Bean Applications via Model Driven Development. In: Proc. ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (2005), Seattle, WA
18. Baude, F., Caromel, D., Huet, F., Mestre, L., Vayssiere, J.: Interactive and Descriptor-based Deployment of Object-Oriented Grid Applications. In: Proc. Of the 11th International Symposium on High Performance Distributed Computing (HPDC'02), Edinburgh, UK
19. Sloane, A.: Modeling Deployment and Configuration of CORBA Systems with UML. In: Proceedings of the 22nd International Conference on Software Engineering (ICSE'00)