Galapagos: Automatically Discovering Application-Data Relationships in Networked Systems

Kostas Magoutis, Murthy Devarakonda, Kiran Muniswamy-Reddy
IBM T. J. Watson Research Center, Hawthorne, NY 10532
{magoutis, mdev} at US dot IBM dot COM, kiran at CS dot HARVARD dot EDU

Abstract: In large networked systems, relationships between applications and the data that they use through multiple tiers of middleware systems are often invisible. While the benefits of knowing such relationships are clear from a systems management perspective, discovery of such relationships is complicated by the widespread adoption of virtualization technologies and the tendency to view each middleware tier as an independent "domain" from a systems management perspective. In this paper we present a methodology and a system for automatic discovery of end-to-end application-data relationships. The key to the methodology is the modeling of data locations from which applications use data and of how middleware systems make data available to software layers above them.

***Keywords-** Component; Networked Systems, Systems Management, Storage Management, Information Management*

I. Introduction

Today large-scale applications follow a tiered architecture and run as large and complex distributed systems consisting of hundreds of hardware and software components [1]. Although there are well known techniques for discovering [1][2][5] and representing [3][4] dependencies between hardware and software components in such systems, there is no previous work on establishing end-to-end relationships between applications and data that they use or vice-versa. The knowledge of such application-data relationships has a number of important benefits: it can be used to derive application-driven information lifecycle management policies, improve the accuracy of root-cause analysis in case of failures, account for storage use on a per-application basis, and establish the desired connectivity between servers and backend storage.

The methodology and the system, called Galapagos, described in this paper is based on the following concepts: 1) Data location and data "export" models of individual software components; 2) Runtime code (mostly scripts) to extract data-specific information from the software components; 3) A distributed crawling algorithm that uses the models and runtime code to collect and build the end-to-end application-data associations.

II. Galapagos OverView

The Galapagos system described in this paper is designed to discover usage of data in a large distributed system. In other words, it enriches basic infrastructure discovery with how data is used by applications (*e.g.*, business objects, tables, files, etc.) in addition to information about data providers (*e.g.*, enterprise information systems, database systems, etc.). Galapagos discovers and represents all end-to-end, multi-tier dependencies between applications and data in an *n*-tiered distributed system. Moreover, it does so in an easily extensible fashion: adding a new (*n+1*th) middleware tier in an *n*-tiered system automatically includes the new tier in its representation of end-to-end relationships. Galapagos does not rely on (but can leverage) active discovery methods; it is thus less intrusive than systems that require them.

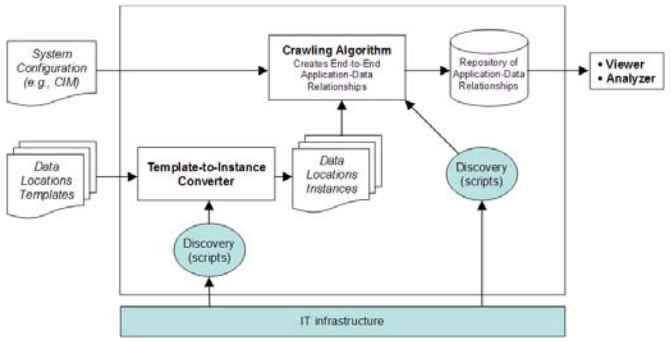

System Configuration (e.g., CIM)
Crawling Algorithm
Creates End-to-End Application-Data Relationships
Repository of Application-Data Relationships
• Viewer
• Analyzer
Data Locations Templates
Template-to-Instance Converter
Data Locations Instances
Discovery (scripts)
Discovery (scripts)
IT infrastructure

Figure 1 Galapagos methodology, architecture

The Galapagos system architecture and methodology Figure 1) is driven by two models, the System Configuration (SC) and a set of Data Locations Templates (DLTs). The SC model describes the IT infrastructure, in a CIM-compliant way [3], and is provided by IT infrastructure discovery systems [4] or manually. A DLT model describes data-specific aspects of a software component, in particular its data locations and data "export" characteristics. Only one DLT is required for each software component type (e.g. one DLT for a DB2 version 9.0). An instance of a DLT is required for each application or middleware component instance in the SC model. Each DLT is extended to a Data Locations Instance (DLI), a superset of the DLT), to capture installation-specific details of the system configuration, such as references to physical or logical assets and pointers to data. DLIs can be automatically derived from the corresponding DLTs by discovering system configuration information through script executions. Following creation of the DLIs, a distributed crawling algorithm performs a traversal of the SC model to generate end-to-end application-data relationships. Starting at the root elements in the SC model (applications and the data they use) and for each application data entity, the algorithm maintains a stack that grows with each visit to an underlying data-providing software component in the SC model. Data entities in a stack are related through data mappings performed by the underlying data-providing software components. The crawling algorithm uses information in the DLIs along the way as well as system information discovered during the crawling process through execution of scripts. Although the core logic of the crawling algorithm is executed at a central location, script execution often requires the invocation of remote management APIs. Where management APIs cannot be remotely invoked, Galapagos agents are required in remote administrative points to exercise those APIs. Automatic installation of these agents (assuming appropriate credentials) reduces intrusion to the managed environment.

III. DLT AND DLI MODELS

DLTs are specified in terms of a meta-model, which is a precise definition of the constructs and rules needed for creating DLTs. The DLT meta-model, which is shown in the UML diagram of Figure 2, describes data consumption and transformation by a software component (application or middleware). A DLT for a specific software component does not contain any installation-specific information. It may, however, contain pointers to *information sources* (*e.g.*, scripts) that can be used at a later time to discover such information. The DLT meta-model consists of two sections: Data Consumption and Data Transformation.

Data consumption of a software component (application or middleware) is expressed as a list of the *datasets* used by the component. Each dataset is associated with a *data type* and a *namespace* whose format is specified in the model. Dataset lists can be either hand-crafted by the creator of the DLT model (if these names do not change across installations of the software components) or can be automatically discovered through the stated information sources. In general, the namespace format used to describe datasets in Galapagos is:

$$\text{PROVIDER} : (\,\text{TYPE}_i\,;\,\text{NAME}_i\,)^{\,i}$$

In the above naming scheme, PROVIDER points to the software component whose DLT describes the data types TYPE_1 through TYPE_i. The index *i* can run from one to a finite number. A dataset name in the above format may also contain variables which are bound at a later time to the output of scripts, as well as wildcards (*e.g.*, the equivalents of *, % in UNIX).

Examples of namespace formats used in the case of (1) a relational database, (2) a file system, (3) an enterprise information system (*e.g.*, SAP) are:

1. dbinstance : dbtype ;dbname / table ; tablename
2. fsinstance : (file-or-directory ; file-or-directory-name) ^{i}
3. eisinstance : repository ; rname / businessobject ; boname

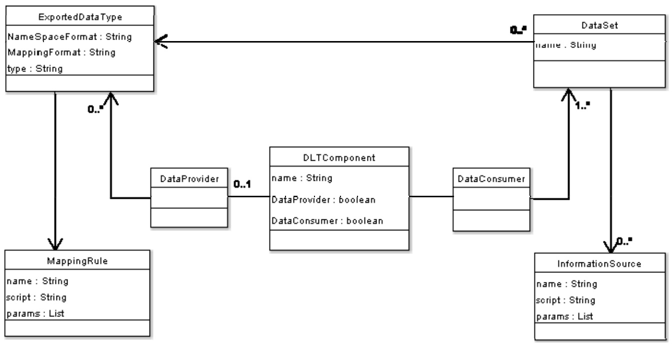


Figure 2 Data Locations Template (DLT) meta-model.

The expression of data consumption is a requirement for all types of software components, whether they are applications or middleware systems. Middleware systems, however, additionally require the expression of the way that they "export" data to software in tiers above them. In Galapagos, we refer to middleware components as *data providers* as they implement and export data abstractions to tiers above them. In addition to defining *exported data types*, data providers also describe corresponding *mappings* between two levels of data abstractions. In general, a data mapping between a (high-level) data abstraction **A** and a (low-level) data abstraction **B** relates data entities between two namespaces, and is represented in DLT as:

$$\text{PROVIDER}^{\mathbf{A}} : (\,\text{TYPE}^{\mathbf{A}}{}_i\,;\,\text{NAME}^{\mathbf{A}}{}_i\,)^{\,i} \rightarrow$$

$$\rightarrow \text{PROVIDER}^{\mathbf{B}} : (\,\text{TYPE}^{\mathbf{B}}{}_i\,;\,\text{NAME}^{\mathbf{B}}{}_i\,)^{\,i}$$

The above expression means that an instance of the high-level data abstraction (A) maps to one or more instances of the low-level data type (B). Details of such a mapping are discovered by executing scripts that invoke middleware management APIs during the crawling phase of the Galapagos system. The expression of the data mapping and the associated dynamic scripts are typically written by middleware developers or experts with intimate knowledge of the particular data mapping mechanics. This is a one-time effort for a given major version of a middleware system, amortized over repeated uses of it by the Galapagos system. In general, the complexity of creating DLTs varies depending on whether the software component represents a simple application or a more complex middleware component. Simpler DLTs can be automatically created by software modeling tools [8].

DLT models do not include installation-specific information. Instead, they contain variables whose values are discovered after software installation and stored in the Data Locations Instance (DLI). Additional information in a DLI includes absolute pathnames of datasets, machine names, and other installation-specific information such as references to instances of installed software and hardware components described in the SC model of the distributed system.

The process of extending DLTs to DLIs uses runtime support such as scripts to mine and extract information from various information sources such as the operating system registries, application server APIs and so on in a distributed system. For example, one way to discover data consumption is by looking at the application container that provides runtime services (*e.g.*, a J2EE application server or an operating system) or application packaging and registry systems (*e.g.*, J2EE .ear/.rar files, Linux RPMs, Windows registry, *etc*.). Once created, DLIs are placed in a centralized repository corresponding to a particular distributed system.

IV. Application-Data Relationship Discovery - the Crawler Algorithm

The following algorithm describes the distributed discovery process used in Galapagos. For simplicity we assume that the lowest data abstraction of interest is a file. We also assume the existence of Galapagos agents on administrative points in the network. Remote procedure calls (RPC) refer to communication with Galapagos agents.

<u>*Inputs*</u>

- System configuration (SC) model
- DLI models for applications and middleware components

<u>*Algorithm*</u>

```
1. For each application $A_r$ in the SC model, consider the
   datasets { $D_i$ } listed in the application's DLI
2. For each such dataset $D_i$, create an empty stack (associated
   with the application $A_r$) and push $D_i$ into it
    2.1 if $D_i$ is a file, record application-file relationship
        (contained in the stack) and backtrack
    2.2 if $D_i$ is not a file and $D_i$ has not been seen before,
        visit the data provider of $D_i$ (represented by a
        node P in the SC model); get a handle on the DLI
        of P
        • use a mapping rule in that DLI to map $D_i$ to a
          list of datasets { $D'_j$ }; the rule may require
          RPC to agent on remote administrative point
        • for each $D'_j$
            • push $D'_j$ to the stack
            • Go to step 2.1 and repeat for $D'_j$
    2.3 if $D_i$ is not a file but $D_i$ has been seen before,
        retrieve relationships between $D_i$ and files and
        add them to stack, then backtrack
```

<u>*Output*</u>: Application-data relationships stored in repository.

The complexity of the Galapagos discovery process is that of depth-first search (DFS) of the SC model graph, multiplied by the number of datasets considered. The cost of visiting each node in the graph depends on the delay of invoking scripts that exercise management APIs associated with the particular node type. For example, accessing the API of a database management system may be a slow process in certain cases. As a result, the overall cost of the Galapagos discovery process could be dominated by the number of such calls (*i.e.*, related to the number of database tables Galapagos needs to resolve).

V. Prototype and evaluation

The current Galapagos prototype is implemented as a stand-alone Java-based system consisting of the following components (Figure 1): Converter of DLT models to DLI models; Crawling Algorithm for discovery of application-data relationships; User interface (UI). The SC model input to Galapagos is provided either by infrastructure discovery systems or composed manually after interviewing system administrators. DLT models are either automatically produced by modeling tools or composed manually by developers or application experts. The overall discovery process, which combines a full SC graph traversal with distributed system infrastructure information, is completed when the data usage of all applications has been drilled down and related to the lowest level of storage hierarchy. End-to-end application-data relationships are stored in a repository (currently a relational database) and retrieved via SQL queries through a command-line interface. Visual inspection of the SC model and application-data relationships is available through an Eclipse-based UI (Figure 3). The discovery process lasts about ten minutes on a system configuration involving two J2EE applications accessing a database of about a thousand tables. Re-discovery can be triggered either periodically or each time installation of a new application or creation of new data is detected.

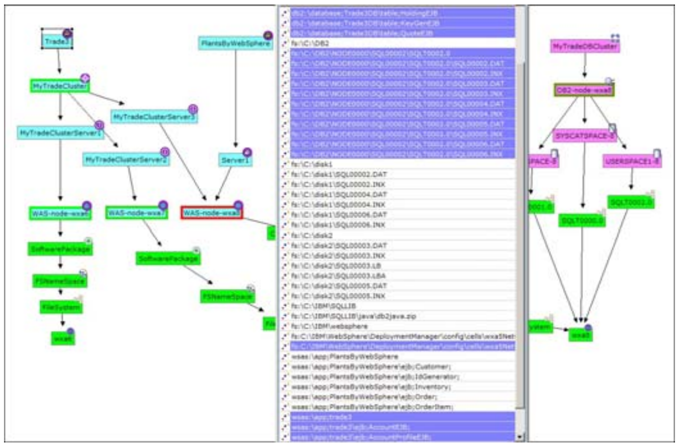
1) Impact of automation on practice of storage administration

The automated discovery and visual representation provided by Galapagos improves over the current state of manual and thus time-consuming and error-prone methodologies. Any systems administrator can testify to the high complexity of manually identifying all data owned by an application. Galapagos however, can automatically discover and provide the list of all datasets (files, tables, etc.) belonging to a particular application (Figure 3). A typical use of end-to-end application-data relationships is in migration of applications across IT infrastructure for the purpose of asset consolidation. In our experimental setup, overall evaluation of all discovered application-data relationships yielded zero "false negatives"-- all files owned by the application and middleware modeled were accounted for-- as well as zero "false positives."

Similarly, Galapagos simplifies the process of identifying the ownership of a dataset. For example, consider the file C:\DB2\NODE0000\SQL00002\SQLT0002.0\SQL00002.DAT stored on a particular computer system. Given the heavily encoded name of the file, it is evident that manual identification even by experts in system administration is difficult and error-prone. A query at the Galapagos repository reveals that the file is managed by DB2, it is part of a table (whose name is QUOTEEJB), and is mapped to a particular J2EE application (Trade3).

2) Challenges

The accuracy and efficiency of the Galapagos approach depends on several factors. First, an important issue is the creation of DLTs for software components, as well as the accuracy and completeness of these models. A DLT that does not fully describe all uses and transformations of data by a software component, will cause some application-data relationships to be missed by the discovery process ("false negatives"). DLT models are easier to create, manually or automatically, in the case of execution environments with well-defined installation and data-access interfaces, such as J2EE, SAP, *etc.* Discovery of data use is harder in less structured application containers. Particularly challenging cases include unstructured applications running "bare-bone" on standard operating systems and placing data in shared directories (*e.g.*, /tmp), shared libraries (*e.g.*, in windows\dll), and so on. Such data use however, will have to be discovered for creating the corresponding DLI model, *e.g.*, by looking at the relevant installation logs.



```
db2:\database:Trade3DB\table:HoldingEJB
db2:\database:Trade3DB\table:KeyGenEJB
db2:\database:Trade3DB\table:QuoteEJB
fs:\C:\DB2
fs:\C:\DB2\NODE0000\SQL00002\SQLT0002.0
fs:\C:\DB2\NODE0000\SQL00002\SQLT0002.0\SQL00002.DAT
fs:\C:\DB2\NODE0000\SQL00002\SQLT0002.0\SQL00002.INX
fs:\C:\DB2\NODE0000\SQL00002\SQLT0002.0\SQL00003.DAT
fs:\C:\DB2\NODE0000\SQL00002\SQLT0002.0\SQL00003.INX
fs:\C:\DB2\NODE0000\SQL00002\SQLT0002.0\SQL00004.DAT
fs:\C:\DB2\NODE0000\SQL00002\SQLT0002.0\SQL00004.INX
fs:\C:\DB2\NODE0000\SQL00002\SQLT0002.0\SQL00005.DAT
fs:\C:\DB2\NODE0000\SQL00002\SQLT0002.0\SQL00005.INX
fs:\C:\DB2\NODE0000\SQL00002\SQLT0002.0\SQL00006.DAT
fs:\C:\DB2\NODE0000\SQL00002\SQLT0002.0\SQL00006.INX
fs:\C:\disk1
fs:\C:\disk1\SQL00002.DAT
fs:\C:\disk1\SQL00002.INX
fs:\C:\disk1\SQL00004.DAT
fs:\C:\disk1\SQL00004.INX
fs:\C:\disk1\SQL00006.DAT
fs:\C:\disk1\SQL00006.INX
fs:\C:\disk2
fs:\C:\disk2\SQL00003.DAT
fs:\C:\disk2\SQL00003.INX
fs:\C:\disk2\SQL00003.LB
fs:\C:\disk2\SQL00003.LBA
fs:\C:\disk2\SQL00005.DAT
fs:\C:\disk2\SQL00005.INX
fs:\C:\IBM\SQLLIB
fs:\C:\IBM\SQLLIB\java\db2java.zip
fs:\C:\IBM\websphere
fs:C:\IBM\WebSphere\DeploymentManager\config\cells\wxa5Net
fs:C:\IBM\WebSphere\DeploymentManager\config\cells\wxa5Net
waas:\app;PlantsByWebSphere
waas:\app;PlantsByWebSphere\ejb;Customer;
waas:\app;PlantsByWebSphere\ejb;IdGenerator;
waas:\app;PlantsByWebSphere\ejb;Inventory;
waas:\app;PlantsByWebSphere\ejb;Order;
waas:\app;PlantsByWebSphere\ejb;OrderItem;
waas:\app;trade3
waas:\app;trade3\ejb;AccountEJB;
waas:\app;trade3\ejb;AccountProfileEJB;
```

Figure 3 Snapshot from Eclipse-based Galapagos UI. Graph depicts SC model, highlighted datasets in the data view belong to a particular application.

Another important challenge is the speed at which Galapagos adapts to new state being created in the distributed system. Examples of new state are the installation of new applications or middleware, the creation of new data by existing applications, *etc.* Two possible choices are to periodically re-start the discovery process, or when possible, initiate it in response to a notification of a change (new application installed, new data created, *etc.*) in the system.

Finally, the fine level of data granularity examined by Galapagos (*e.g.*, files) raises scalability issues. The use of *summarization* of filenames (mapping all files with the same ownership under a given directory to the pathname of that directory) is one way to address these issues. For example, all files under C:\Program Files\DB2\SQLLIB\ belong to DB2 and thus need not occupy more than a single row in the Galapagos repository.

VI. Related Work

Previous studies of discovering dependences between distributed systems tiers using online system monitoring of network traffic and statistical heuristics [1] are potentially applicable to discovering application-data relationships. However, such systems generally have several drawbacks: (a) being based purely on heuristic rules, they cannot eliminate the possibility of missing some application-data relationships ("false negatives"); (b) they can not be generalized easily to multi-tiered distributed systems. We believe however, that a heuristic approach is useful (particularly when modeling information is not available) and is thus complementary to the approach described in this paper.

Various systems have investigated building distributed system dependency graphs using passive (*e.g.*, trace collection and offline analysis [1]) or active (*e.g.*, fault injection [5]) methods. Some of the uses of a dependency graph include problem determination, performance analysis, and visualization. Galapagos relates to these approaches in that it also focuses on discovering dependency information; however, it differs in that it expresses dependency specifically as it relates to applications' use of data, which has a finer grain scope than dependency between software components. Systems tracing the provenance of data [6][7] are also related to our work in that they establish a history of changes to data, and the history may include the applications that made the changes. However, the provenance concept is evolving and distributed multi-tiered systems are way beyond the scope of present provenance prototypes.

VII. Summary

In this paper we presented a novel approach to automatically discovering application-data relationships in multi-tiered distributed systems, based on modeling the consumption and transformation of data by software components. We showed that our models are general enough to encompass a wide range of middleware systems and applications. Intellectual energy is invested once (typically by the application developers) to create these models and then the models are used again and again (in every instance of the application deployment) to leverage the investment. We described a distributed crawling algorithm that uses the model information to automatically construct application-data relationships.

References

[1] M. Aguilera, J. Mogul, J. Wiener, P. Reynolds, A. Muthitacharoen, "*Performance Debugging for Distributed Systems of Black Boxes*", in Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003), Lake George, NY, October 2003.

[2] V. Machiraju, M. Dekhil, K. Wurster, J. Holland, M. Griss, P. Garg, "Towards Generic Application Auto-Discovery", HP Labs Technical Report 1999-80.

[3] Common Information Model (CIM), Data Management Task Force (DMTF), http://www.dmtf.org/standards/cim/

[4] Configuration Management Database (CMDB), IT Infrastructure Library (ITIL), http://www.infra.com.au/Solutions/ConfigurationMgnt.asp

[5] A. Brown, G. Kar, A. Keller, "*An Active Approach to Characterizing Dynamic Dependencies for Problem Determination in a Distributed Environment*", in Proc. of the Seventh IFIP/IEEE International Symposium on Integrated Network Management (IM 2001), Seattle, WA, May 2001.

[6] K. Muniswamy-Reddy, D. Holland, U. Braun, M. Seltzer, "*Provenance-Aware Storage Systems*", in Proc. of the 2006 USENIX Annual Technical Conference, Boston, MA, June 2006.

[7] P. Groth, M. Luck, L. Moreau, "A Protocol for Recording Provenance in Service-Oriented Grids", in Proc. 8th Int. Conf. on Principles of Distributed Systems, December 2004, Grenoble, France.

[8] IBM Rational; Unified Modeling Language, http://www-306.ibm.com/software/rational/uml/