DADL: Distributed Application Description Language

Jelena Mirkovic, Ted Faber, Paul Hsieh
USC Information Sciences Institute
sunshine, faber@isi.edu, hsieh@usc.edu

Ganesan Malaiyandisamy, Rashi Malaviya
Infosys Corporation
Ganesan_M, Rashi_Malaviya@infosys.com

Abstract

Cloud computing infrastructures enable enterprises to acquire equipment and pay associated costs only when their business demands this, which is very attractive. However, there are notable challenges to wide use of clouds for enterprise services, such as security and privacy of data stored in clouds and reliability of cloud services.

Our research focuses on two subproblems in this space. First, we believe that the future of cloud computing lies in cloud specialization and differentiation. Future businesses will thus be allocating resources from different clouds, some public, some private, for a single application. Our research investigates how to perform automated, optimal allocation of these resources starting from (1) a description of distributed application components' needs, and (2) a description of available cloud resources. Second, our research investigates how to offer multi-layer reliability monitoring in a manner that is application-transparent, platform-transparent, and real-time.

This paper presents a language, called DADL (distributed application description language), that can be used to express (1) architecture, behavior and needs of a distributed application that may be deployed on clouds; and (2) specifics of available cloud resources. In our future work, this language will become input to our automated allocation and reliability monitoring systems. DADL is implemented as an extension of SmartFrog [8] – a framework for configuring, deploying and managing distributed software systems.

1 Introduction

Cloud computing infrastructures offer unprecedented advantages to businesses. They enable enterprises to scale their resources up or down with demand, expanding the business and following market trends closely, while saving on equipment purchase and maintenance. Unfortunately, cloud services provide little reliability or security guarantees. This makes them ill-aligned with businesses that often must offer such guarantees to their customers, and leads to a trend where some business application components run on private clouds, where confidentiality and privacy can be guaranteed, and others on public clouds, where resources are cheap.

We believe that this trend of application distribution on multiple infrastructures will increase with proliferation of cloud infrastructures and their specialization. Future businesses will likely acquire resources from multiple clouds to meet unique needs of their applications, minimize cost and maximize security and reliability. Our research focuses on two services to support this future trend: (1) automated allocation of resources from multiple cloud infrastructures based on a description of application architecture and needs, and a description of available cloud resources, and (2) multi-level reliability monitoring, problem detection and recovery.

Our first step toward developing these services was to develop a language that can be used to specify (1) application architecture, behavior and needs with regard to confidentiality, reliability, special resources, etc. and (2) cloud resources, their features, availability and cost. This paper describes this language, called DADL: Distributed Application Description Language, and its use.

2 Application Model

We define an *application need* as a need for a special resource (e.g., 1TB storage, 1Gbps bandwidth), service (e.g., auto-scaling offered by Amazons EC2), feature (e.g. confidentiality) or performance target (e.g., request/response time below 1 second). We pay special attention to confidentiality requirement, and define this as requirement to minimize the possibility of proprietary data exposure, either through unprotected inter-resource communication or through placement of data on a public resource.

We define *reliability* as providing a continuously good service quality to enterprise clients in face of failures or, if that is not possible, fast failure detection and recovery. Qualitative attributes ”good” and ”fast” in this defi-

nition must be replaced by quantitative attributes by the application developer, and are application and enterprise specific.

Our application model, presented in the rest of this Section, defines application components, their needs, and their dependencies. DADL language constructs are built on top of that model.

2.1 Application Components and Needs

We define three types of application components: `Elements`, `Groups` and `Channels`.

A simple application *`Element`* represents a basic functional unit in the application such as a database server, a storage server, etc.

A *`Channel`* describes a virtual communication link between application `Elements` and/or `Groups`. Once the application is deployed, it may consist of a series of physical links, within and across clouds requiring authentication negotiation. `Elements` and `Channels` that connect them can further be grouped together into compound `Elements`, for easier manipulation.

`Groups` combine `Elements` and `Channels` into a single scalability unit. Their attributes define when additional physical resources should be allocated to the `Group`, to scale up its performance or improve reliability, and how to migrate data as the `Group` size grows and shrinks. Our aim is that this information should support automated scaling of an application on clouds. A `Group` is regarded as a single compound application `Element` by other `Elements`, thus, application architecture remains the same regardless of physical size of the `Group`. Further, all `Elements` within the same `Group` are implicitly connected, supporting any data migration pattern.

A component *need* describes performance, security, reliability and any special resource requirements that application `Elements`, `Groups` and `Channels` must meet for satisfactory performance of the distributed application.

2.2 Component Dependency

Distributed applications, especially those deployed in clouds, require automated initialization sequence to achieve scalability and dynamic reconfiguration, and to minimize errors. The initialization and shutdown dependencies between components are thus part of our model.

3 DADL

Distributed Application Description Language (DADL) is based on the model presented in the previous Sections. DADL describes application components, `Channels` and `Groups` and their respective needs. It also describes available cloud resources and their features. We envision that the application description will be written by application designers in a user-friendly GUI, while the resource specification will be automatically extracted from cloud infrastructures by our programs. These two descriptions are inputs to our future automated allocation and reliability monitoring systems.

3.1 Candidate Languages

While constructing DADL from scratch was possible, extending an existing language had significant advantages. First, we wanted to leverage some features of existing languages, such as an existing interpreter, an ability to automate application deployment or a presence of a well developed GUI for language specification. Second, extending an existing, well-used language improved chances of DADL adoption by users that are already familiar with the basic language.

We have considered the following possible candidates for extension: Durra [6, 5], XML [1], Ruby [2], UML [4] and SmartFrog [8].

Similarly to DADL, Durra [6, 5] is a language for distributed application description. It is customized for a class of real-time, embedded applications, such as sensor data collection, that concurrently execute multiple tasks on heterogeneous resources and communicate via message passing. This focus on process description makes extending Durra to describe cloud computing applications somewhat awkward. Durra further lacks the notion of a "need" that is critical for DADL, and adding this to Durra would be a significant change to its architecture.

XML [1], Ruby [2] and [4] are popular languages that are general-purpose, and DADL constructs could be built on top of them. Using these languages would give us an advantage of well-developed editors, interpreters/parsers and a wide user base. On the other hand, because these languages are general-purpose we would have to introduce the notion of a distributed application to them, and provide necessary services such as automated deployment. This would be a significant departure from their original usage modes, making costs outweigh the benefits.

SmartFrog [8] is a Java-based framework for configuring, deploying and managing distributed software systems. It consists of a language, a runtime system and a library. SmartFrog language defines configurations of applications. The runtime system provides functionality of deploying application components, and managing running applications based on this configuration. The library implements SmartFrog component model, and some other services. SmartFrog's focus on distributed systems and a framework for their automated deployment made it an attractive candidate for DADL exten-

```
Element extends Prim {}
Group extends Element {}
Channel extends Prim {}
```

Figure 1: Application components.

sions. Its constructs needed minimal extensions to express our application model that are introduced via Java's inheritance model. We further benefited from its mature, open-source code and its readable attribute-value pair representation. Application needs were added to SmartFrog by specifying new attributes. SmartFrog further defines component dependencies, which influence their deployment order, and this is well aligned with our notion of dependency in DADL.

3.2 DADL Extensions to SmartFrog

We now present the SmartFrog extensions we created to support our application model, and to describe cloud resources.

3.2.1 Component Extensions

We define application `Element`, `Group` and `Channel` in SmartFrog as shown in Figure 1. An `Element` represents a single component and its attributes describe this component's needs. It extends a generic class in SmartFrog called `Prim` that all classes extend. A `Group` binds together those `Elements` that can scale elastically and automatically with the load. In addition to `Element` attributes, a `Group` has group attributes that describe triggers for scaling up and down, how that scaling occurs, and how data is migrated to new members. A `Channel` describes connections between `Elements` or `Groups`. All components in DADL extend one of these three building blocks.

3.2.2 Need Extensions

In this Section we present DADL extensions to SmartFrog that can be used to describe those application needs that we believe are common to most applications. They are summarized in Figure 2. We do not claim that this set is complete, but we believe this is a basic set, useful for many applications. DADL can be easily extended to express other application needs as they become necessary.

- Architecture: `Element`'s optional attributes `osImage` (string, specifying the image name) and `CPUArch` (string, specifying the architecture name) express application needs for a specific operating system or CPU architecture (e.g. 32-bit or 64-bit, i386, etc.).

```
Element attributes:
  /* architecture */
  osImage          // string
  CPUArch          // string
  /* processing capability */
  minCPUSpeed      // float, unit GHz
  minCPUCores      // int, >= 1
  /* memory */
  minMemory        // float, unit GB
  /* storage */
  minDiskCapacity  // float, unit GB
  /* confidentiality */
  isConfidential   // binary

Group attributes:
  /* scalability */
  loadUp           // float, 0-1
  loadDown         // float, 0-1
  unitSize         // float, >0
  repAction        // string
  minReplicas      // int, >= 1

Channel attributes:
  /* connectivity */
  from             // string
  to               // string
  /* network */
  maxDelay         // int, unit ms
  minThroughput    // float, unit Mbps
```

Figure 2: Application needs.

- Processing capability: Computing-intensive applications often require some minimum processing power to achieve performance targets. Optional attributes `minCPUSpeed` (float, unit GHz) and `minCPUCores` (positive integer) express these needs in the `Element` component.

- Memory: An application may keep a lot of data in memory, and thus may require some minimum memory size to achieve target performance. `Element`'s optional attribute `minMemory` (float, unit GB) expresses this need.

- Storage: An application may need a lot of disk space to store large inputs, outputs or intermediate results. `Element`'s optional attribute `minDiskCapacity` (float, unit GB) expresses this need.

- Network: Many distributed applications depend on the network resource for reliable operation. They communicate data among their components and between their components and the outside world (application clients, public Internet servers, etc.). `Channel`'s optional attributes `maxDelay` (integer, unit ms) and `minThroughput` (float, unit Mbps) are used to describe an application's network needs. Required attributes `to` and `from` describe which components the `Channel` connects. These could be other application components, the IP address or URL of an outside server, or the reserved key word `client` denoting connection to the application client.

```
CloudInfo extends Prim {}
ResourceInfo extends Prim {}
MachineInfo extends ResourceInfo {}
```

Figure 3: Resource component.

- Scalability: Distributed applications may gain performance boost when their execution is parallelized. As business demand fluctuates, clouds offer special appeal to enterprises by allowing them to scale up or down their resources to follow these fluctuations. In DADL the mandatory `Group` component's attributes `loadUp` and `loadDown` (floats, between 0 and 1) describe thresholds for up and down scaling, the mandatory attribute `unitSize` (positive float) describes the scale of the replication/reduction (e.g. value 2 would mean that the group size doubles when load is too high and halves when load is too low), and the optional attribute `repAction` (string) points to the user-supplied script that should be run to keep the data consistent when group size changes. The optional attribute `minReplicas` (positive integer) gives the minimum size of the group, and is set to 1 by default.

- Confidentiality: Many business applications have proprietary data that cannot be placed on public clouds because its security cannot be guaranteed. The optional `Element` attribute `isConfidential` (binary, 0 or 1) expresses this need.

SmartFrog supports specification of component dependencies and we leverage this for our purpose. It further has a lifecycle mechanism that manages states of components and transitions of states. This makes application initialization and termination occur in the desired partial order. We plan to leverage these features in our future work to deploy application components on resources we obtain via our automated allocation.

3.2.3 Resource Extensions

DADL also encodes available resource information that is automatically extracted from cloud infrastructures. This section presents SmartFrog extensions to express this information. Our future automated allocation mechanism will use resource availability information to determine how application components should be deployed on available resources to satisfy the needs given in the application description.

Three additional DADL components are defined for resource description as shown in Figure 3. The `CloudInfo` contains availability information for an entire cloud infrastructure. It consists of one or multiple `ResourceInfo` components that describe available resources per resource type. The `MachineInfo` component inherits from `ResourceInfo` and describes PCs, which are commonly offered resources in clouds. Other resources can be similarly defined in DADL, such as secure storage, a replicated DB, etc.

```
CloudInfo attributes:
  /* ownership */
  private        // binary

ResourceInfo attributes:
  /* general */
  available      // int, >0
  rentPeriod     // float, unit hours
  cost           // float, $ per rent period
  history        // string

MachineInfo attributes:
  /* hardware */
  arch           // string
  CPUSpeed       // float, unit GHz
  CPUCores       // int, >0
  memory         // float, unit GB
  diskCapacity   // float, unit GB
```

Figure 4: Resource attributes.

Attributes for new resource components are summarized in Figure 4. The mandatory `private` attribute (binary) in `CloudInfo` expresses if the cloud is private or public. The `ResourceInfo` component may contain the `available` attribute (positive integer) that specifies how many resources of a given type are available in the cloud infrastructure. Some clouds reveal this information but most do not. The attributes `rentPeriod` and `cost` (both floats) are mandatory and express the dollar cost of renting that resource for the given number of hours. The `history` attribute (string) is optional and it points to the file containing reliability information collected for the past allocations of this resource type.

4 A DADL Example

In this Section we illustrate how DADL would be used to describe an example application and a cloud infrastructure.

4.1 Application Description: Web Service

The Web service in our example has three components: a set of of workers, a centralized database and a load balancer. Workers process service requests and retrieve/store data from/to the database when needed. The load balancer accepts service requests, dispatches requests to workers and sends results from workers back as responses to service clients.

While this Web service is simple, it has basic application needs, which exist in many business applications:

```
Worker extends Group {
  minCPUSpeed 2;         // 2 GHz
  minMemory 2;           // 2 GB
  loadUp 0.7;
  loadDown 0.3;
  unitSize 2;
  minReplicas 3;
}

Database extends Element {
  isConfidential 1;
}

LoadBalancer extends Element {
}

LBChannel extends Channel {
  from LoadBalancer;
  to client;
  minThroughput 10;      // 10 Mbps
  maxDelay 100;          // 100 ms
}

LBWChannel extends Channel {
  from LoadBalancer;
  to Worker;
  maxDelay 10;           // 10 ms
}

WDBChannel extends Channel {
  from Worker;
  to Database;
  maxDelay 10;           // 10 ms
}

WebService extends Compound {
  SLB extends LoadBalancer;
  SDB extends Database;
  SW extends Worker;
  LBC extends LBChannel;
  LBWC extends LBWChannel;
  WDBC extends WDBChannel;
}
```

Figure 6: Web service description in DADL.

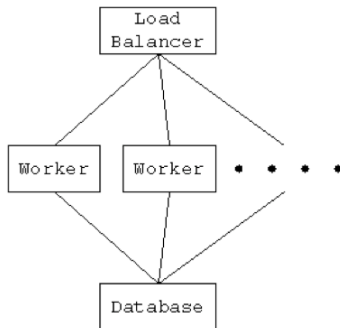

Figure 5: Web service architecture.

- Some components, in our case workers, have performance needs. They need to respond quickly to users and handle multiple requests simultaneously.
- Some components, in our case the database, contain proprietary data and must be deployed on private, secure resources.
- Interconnections between components must be able to support the load and achieve desired response time to user.
- For some components, in our case workers, the number of allocated machines many vary dynamically to scale up with the load.

Figures 5 and 6 show the architecture of the Web service and its specification in DADL. The `Worker` component extends `Group`, because it can scale with the load, and has both `Element` and `Group` attributes. The `Element` attributes describe the processing speed and memory needs, while the `Group` attribute describes that

```
DETER extends CloudInfo {

  private 0;               // public cloud

  pc3000 extends MachineInfo {
    /* hardware */
    arch "i386";
    CPUSpeed 3;            // 3 GHz
    CPUCores 1;
    memory 2;              // 20 GB
    diskCapacity 36;       // 36 GB
    /* general */
    rentPeriod 1;
    cost 0;                // free
    available 49;          // 49 nodes available
  }

  pc2133 extends MachineInfo {
    /* hardware */
    arch "i386";
    CPUSpeed 2.1;          // 2.1 GHz
    CPUCores 1;
    memory 4;              // 4 GB
    diskCapacity 250;      // 250 GB
    /* general */
    rentPeriod 1;
    cost 0;                // free
    available 9;           // 49 nodes available
  }
}
```

Figure 7: Cloud (DETER) description in DADL.

the allocation should double when the average load is above 0.7, and halve when the average load is below 0.3. There are no consistency actions to be taken when group size changes. The minimum size of the group is 3 workers. The `Worker` also needs a fast connection to the database, which is described with the `WDBChannel` component.

The `Database` component specifies need for storage and confidentiality. The `LoadBalancer` com-

ponent describes its need for minimum throughput and maximum delay from/to the client using the `LBChannel` component, the minimum delay to/from workers using the `LBWChannel` component. Finally, the `WebService` component contains the `Worker`, `DataBase` and `LoadBalancer` components.

4.2 Resource Description: DETER

We have developed an automated script to extract resource type and availability information from the private cloud we run at USC/ISI. This is the DETER testbed for cyber-security experimentation [7], where users obtain physical machines for their exclusive use. Figure 7 shows a part of resource availability information extracted by our script. It shows that there are 49 machines of type pc3000 and 9 of type pc2133 available at the time. It also specifies that DETER is a public testbed and it is free to use. Our future work will extend this script to mine necessary information from other clouds.

5 Conclusions and Future Work

As cloud computing infrastructures become more attractive to enterprises, this attractiveness is dampened by the lack of security and reliability guarantees they can offer. At the same time cloud infrastructures continue to proliferate and diversify. This opens an interesting opportunity to address the reliability challenge by keeping up to date and historical reliability information and using it as input in resource allocation. At the smallest sign of trouble resources could be released and other allocated from a more reliable infrastructure. Similarly, the security challenge can be addressed by keeping the sensitive code and data in a private cloud and outsourcing the rest of the application components to cheap and scalable public clouds. Distributing application components over multiple clouds for security, diversity and reliability is an attractive choice that our research explores.

In this paper we have proposed a distributed application description language, called DADL, that can be used by developers to describe application components, their dependencies and needs. We have also developed automated scripts that mine resource availability information from clouds. DADL has constructs to encode this information as well. DADL is built as an extension to SmartFrog language – we have detailed the proposed extensions and illustrated them on an application and resource description example.

Our future work will start from DADL descriptions of application needs and available resources and develop an optimal resource allocation algorithm. This algorithm must select resources that meet application needs, maximize reliability and minimize cost. Some application component attributes match static resource attributes directly and it can be easily seen how these can be combined by the algorithm to achieve allocation goals. Other attributes, such as `maxDelay` and `minThroughput` in `Channel` must be carefully matched with the reliability history in `ResourceInfo`, and different matching approaches will land on different points on the performance/cost tradeoff curve.

Another effort we plan to undertake is to develop an application-independent, multi-level resource monitoring approach. The approach will integrate basic node statistics, such as CPU load, disk usage, etc. with application-level statistics mined from application logs and with network statistics obtained through periodic probing. These will be available in real time to users, they will be stored in resource history files to drive future allocations, and they will also be used to detect potential problems and reallocate resources.

DADL attempts to define a standard vocabulary for application and cloud resource description. What we have presented in this paper will work for some popular applications and cloud platforms but not for all. Investigating necessary extensions to DADL, and how to achieve completeness, will be part of our future work.

Finally, we would like to evaluate our work on real use cases from enterprises, such as Infosys.

6 Acknowledgments

We are grateful for the Infosys Corporation's support of this work via contract number ITL/0208. Opinions expressed in this paper are authors' only.

References

[1] Extensible Markup Language (XML). `http://www.w3.org/XML/`.

[2] Ruby. `http://www.ruby-lang.org/`.

[3] SmartFrog Open Source Website. `http://www.smartfrog.org/`.

[4] Unified Modeling Language (UML). `http://www.uml.org/`.

[5] BARBACCI, M. R., WEINSTOCK, C. B., DOUBLEDAY, D. L., GARDNER, M. J., AND LICHOTA, R. W. Durra: a structure description language for developing distributed applications. *Software Engineering Journal* (1990).

[6] BARBACCI, M. R., AND WING, J. M. A language for distributed applications. In *Proceedings of the International Conference on Computer Languages* (1990).

[7] BENZEL, T., BRADEN, B., FABER, T., MIRKOVIC, J., SCHWAB, S., SOLLINS, K., AND WROCLAWSKI, J. Current Developments in DETER Cybersecurity Testbed Technology. In *Proceedings of the Cybersecurity Applications and Technology Conference For Homeland Security* (2009).

[8] GOLDSACK, P., GUIJARRO, J., LOUGHRAN, S., COLES, A., FARRELL, A., LAIN, A., MURRAY, P., AND TOFT, P. The SmartFrog configuration management framework. *ACM SIGOPS Operating Systems Review* (2009).