Specification and Architecture Supports for Component Adaptations on Distributed Environments *

Chung-Kai Chen, Cheng-Wei Chen, Jenq-Kuen Lee
Department of Computer Science
National Tsing Hua University
Hsinchu 30043, Taiwan
{ckchen, cwchen}@pllab.cs.nthu.edu.tw and jklee@cs.nthu.edu.tw

Abstract

With the arrival of the new computing paradigm in addressing autonomous systems for heterogeneous distributed architectures, distributed component technologies face challenges ahead. One of the key issues will be how one can have component models adapt and respond to environment changes autonomously. In our research work, we argue that additional annotation specifications for components are needed to advance this process. In this paper, we first present additional annotation specifications for components. This information can then be retrieved by Java introspection and represented in DAML+OIL language, which is based on the RDF schema and the XML syntax. Based on the specifications, we then present a Component Management Service (CMS) model and architecture to address the specification and composition issues of components on heterogeneous distributed architectures. Experimental results show significant performance improvements with our support of component adaptations in all cases. Our work presents a major advance for areas related to specifications and compositions of distributed components.

1. Introduction

Component software technologies and architectures have grown with great popularity along with the advances of web and internet technologies. The successes are coming from several compound factors. A try in the analysis of these factors might include abstractions of object-oriented concepts which separates the implementation from interfaces, open and distributed component architectures which enables building of cooperative applications, reusable components which allows managements of software complex, the possibilities of component compositions at runtime which makes software more flexible and extensible. While enjoying great successes, component technologies are with great challenges ahead. This is in part due to an era of the new arrival of computer architectures including the distributed enterprise architectures built with individual components linked by messaging subsystems, the autonomous [8] and grid-like architectures [10] in the web, and the distributed and pervasive architectures assembled by a group of small devices and connected by heterogeneous wireless environments.

We argue that additional properties of component software need to be annotated in addition to the separation of implementations from interfaces. We think this is important for component software and architectures to meet the new era of distributed systems. We give the issues below. First, component architectures and designs introduce the concept of abstraction, where the interface specification of an object is separate from the implementation. This will allow applications to use the components without detailed and internal knowledges of implementations. With the arrival of autonomous computing systems and availability of service compositions in the web, applications are eager to query the properties of the component while compose or re-use components. These properties might include the Quality of Service assured, performance parameters when used in heterogeneous network environments, system requirements when used in small devices, etc. These properties represent a new dimension of specifications for components. Second, when used in a distributed architectures assembled by small devices and connected by heterogeneous wireless environments, specifications on software components will allow the adoption of components best adaptive to given en-

*This work was supported in part by NSC under grant No. NSC92-2213-E-007-054 and NSC92-2213-E-007-007, MOE research excellence project under grant No. 89-E-FA04-1-4, MOEA research project under grant No. 92-EC-17-A-03-S1-0002, and NTHU-ITRI research project under grant No. 92-EC-17-A-99-R1-0337. The component-related technology of this work was also supported in part by Microsoft Research Innovation Excellence Awards for Embedded Systems, 2003.

vironments. Due to object-oriented techniques, the component resources have common information associated with them, the interfaces they implement. We argue that additional descriptions can be added. For example, a MP3 decoding component may have the version number information, the computing power needed for the lowest sound quality acceptable, the memory footprint of its implementation, and the license about its usage. These properties, we called metadata, address the difference between the implementations of the interface. It helps us search not only the kind of components, but also specific realizations.

In this paper, we present a Component Management Service (CMS) model and architecture to address the specification and composition issues of components on distributed and parallel environments. There are three key elements in CMS systems. First, we annotate additional specifications for components. Our system is currently mainly based on JavaBean programs, and we propose an ontology specification for JavaBean programs to attempt to add specifications for JavaBean programs. This additional interfaces for components can then be retrieved by Java introspection and represented in DAML+OIL language, which is based on the RDF schema and the XML syntax. An algorithm is also presented in transforming annotation interfaces of components into the format of DAML+OIL language. The vocabulary of this ontology provides a basic terminology to annotate components with the information about the conditions and suggestions of adopting a component for component specializations. It also gives a reference criteria for choosing the most suitable components at a given time and environment for performances and functionality purposes. With our design of the annotations, it's also possible to automatically retrieve the annotations of object components, connect them by their object-oriented relationships, organize them to form CMS databases, and discover them in the CMS databases by component characteristics. The second element of the CMS system is the support for component compositions. As our system is currently mainly based on JavaBean programs. We are employing Java proxy and object serialization mechanisms for supporting compositions of components. We also classify components into sessionless and sessionful objects, and give the definitions of composition points of programs, respectively. Finally, the CMS architecture also supports the compositions of a group of components related to each other on distributed environments. For example, the component replacements of encoder and decoder distributed on different sites will need both of the components replaced, together. We present algorithms and protocols for component replacements in this scenario.

The employment of runtime compositions of components and the ontology specification proposed in our CMS system enable programs to adapt objects dynamically according to application and architecture characteristics. We demonstrate several application scenarios via experiments. First, We demonstrate experiments with matrix computations on clustered environments. A CFD code for the parallelization of pressure correction method is used. In the application, it will need matrix component for a conjugate gradient solver. Initially, applications executed with dense component matrix. At runtime, if given inputs of dataset are very sparse, matrix components equipped with ontology specifications can be replaced by sparse matrix components. In the case datasets are dense but it takes a long time to finish, a parallel dense matrix component can also be employed with our CMS architectures. Second, we demonstrate experiments with component applications running with Java RMI on cluster environments with heterogeneous network environments. RMI components can be adapted from ethernets to VIA or InfiniBand environments with different RMI component implementations and specification supports in our CMS architectures. Finally, we also demonstrate Java Grande benchmark applications running on heterogeneous wireless environments. In the runtime, Java RMI objects running over GPRS wireless network can be adapted to RMI objects over WLAN environments if such environment is available. Again this adaption can be done with the help of component specifications. Experimental results show significant performance improvements with our support of component adaptations in all cases. This work is a part of our research efforts in supporting high-performance object-oriented environments on heterogeneous network environments and under university research excellence project.

The remainder of this paper is organized as follows. Section 2 presents the component annotations. Next, Section 3 gives component management service models and architectures. Section 4 then presents experimental results. Next, Section 5 gives related work and Section 6 concludes this paper. Finally, Appendix gives the translation algorithm from annotation interfaces to DAML+OIL languages.

2. Component Specification and Annotation Ontology

One of the most important promises of object-oriented technology is the increase of code reusability. Components can be reused in different applications as long as they adhere to the same interfaces. Similarly, applications can also adopt any component as long as it meets/implements the expected interfaces. Implementation details are hidden below the interfaces, thus we gain the good possibility of reuse.

When there are more than one choice to be adopted by an application, we will require more "detailed" information about the implementations in order to make a good choice. We will need a mechanism to annotate/inspect a compo-

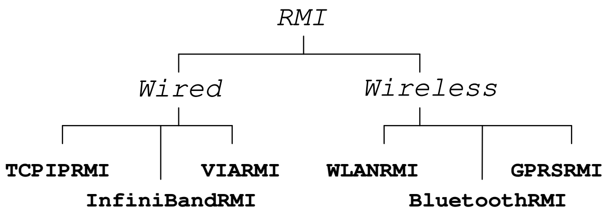

Figure 1. The family of possible RMI implementations.

nent for automaton. In this section, we proposes a mechanism called *annotation interfaces* to serve as the annotation of components. Through the use of annotation interfaces a component writer can even embed dynamic information into components. Another important issue addressed here is the recognition of the appropriateness of a component adoption. We need an systematic approach to judge the appropriateness of the component implementations currently used or judiciously select the ones that possibly will benefit the systems. We define an ontology to formalize the knowledge the component interfaces present, after analysis on components. Based on this knowledge, the system can make judgments on component selections in response to the changes of environments.

We describe the techniques here with a running example, the RMI components. Another example regarding to matrix components is given in [4] and will also be used later as one of our experiments. RMI is a standard way for Java programs to invoke object's methods remotely. The transport layer of RMI can be abstracted and represented as an object in a clear implementation [16]. For the simplicity of description, we assume that the interface of the transport objects is called `RMI` and there is a family of these transport layer objects regarding to different network media, as Figure 1 shows. We will show how these components can be annotated with our method in the context in each subsection below.

2.1. Annotation Interface

Our specification mechanism is to ask component library designers to write additional annotation interfaces on top of component programs. Annotation interfaces are the interfaces in Java that a component implements additionally for revealing its own implementation-related characteristics [4]. By calling the methods of annotation interfaces on a component, we get these metadata returned. Thus the programmer can even provide dynamic information depending on the runtime environment with complex logics. This makes the specification as an executable logics. In our case here, we need annotate additional interfaces for RMI components to specify performance and environment parameters.

We classify the annotation into four catalogs as described below. We can use naming convention rules of the annotation interface to classify the class of annotation methods.

- **Inspection** denotes methods to inspect the investigable runtime characteristics of components. It represents logics to fetch the implementation metadata of the component that are static (such as version number), loaded, or linked dynamically in the runtime, possibly from an XML file or an information server. It can also define profiling methods that will be provided by the component writers to perform implementation-specific profiling.
- **Requirement** presents quantitative requirements on properties. Every method in it has the name started with "getReq" and ended with "UB", "LB", "CON" or "EQ", indicating whether the upper bound, lower bound, subset, or value requirement. The remaining part of the method name is the name of the property.
- **Preference** presents quantitative preferences on properties. Every method in it has the name start with "getPref" and end with "BT", "ST" or "APP", indicating whether the preferred value is as big as possible, as small as possible, or as close to a given value as possible. The suggested value of the denoted property is given with the function invocation to methods of this family.
- **ExternalRef** returns a reference to external components. This can point to the inspection methods in other components such as environmental components. When a component in execution relies on the environmental facts or other components, we use this to refer to components (such as environment component) for query of the component states.

Note that the annotation mechanism for the required/preferred condition can be dynamic. For example, an image processing component may prefer a certain amount of memory in order to have acceptable performance. The memory size it prefers might depend on the average size of images in its current work-chunk instead of a fixed preset value.

Figure 2 shows `VIARMI` and the possible annotation interfaces for it. This is a RMI component based on VIA implementations and with annotation interfaces.

The `VIAAnno` class is the annotation interface of `VIARMI` and encompasses the access methods of VIA-related characteristics. The evaluation of the suitability of a component is based on the quantitative values returned by those methods in the annotation interface. The values from `getExtViplVersion`,

```
public interface VIAAnno extends Annotation {
    public String getReqViplVersionLB();
    public float getPrefAveragePacketSizeST();
    public float getPrefAverageBandwidthBT();
    public float getPrefAverageLatencyST();
    public String getExtViplVersion();
    public float getInspAveragePacketSize();
    public float getInspAverageBandwidth();
    public float getInspAverageLatency();
}
public class VIARMI implements RMI, VIAAnno {
    ...
}
```

Figure 2. `VIARMI` **and the annotation interfaces for it.**

`gettInspAveragePacketSize`, `getInspAverageBandwidth` and `getInspAverageLatency` will be compared with values from `getReqViplVersionLB`, `getPrefAveragePacketSizeST`, `getPrefAverageBandwidthBT` and `getPrefAverageLatencyST`. The merit of using annotation interfaces is that the annotation form is not attached static data but program logics revealing dynamic data. The component writer can provide runtime information about his implementation to the component user.

2.2. The Ontology for Annotations

After we implement the annotation interfaces with logics, the characteristics of a component can then be automatically generated in the runtime by using the Java Reflection APIs and the help of naming conventions. This description contains information about the syntactical structures, the supported annotations, and the inspected values of runtime characteristics of components. The values presented in this description is the snapshot result of the component at the moment of generating.

In order to exchange and process this information in a formal way, we have defined an ontology that provides a standard set of vocabularies in presenting the descriptions above. The ontology is written in DAML+OIL languages. This definition is maintained online [5, 4] for further access. We summarize the key vocabularies below:

- the **FunctionItf** denotes those interfaces that the component originally implements. The **hasExtension** denotes the extension relation from a function interface to its extensions.
- the **hasImplementation** and **implements** denote the inheritance relationship between the function interfaces and the component, which is denoted by **Implementation**.
- The four kinds of annotation are denoted by **Inspection**, **Requirement**, **Preference**, and **ExternalRef**.
- The relation from components to annotations is denoted by **withAnnotation**.
- The preference/requirement relationships are denoted by **smallerThan**, **biggerThan**, **approximating**, **lowerBound**, **upperBound**, **withValue** and **contains**.

Figure 3 illustrates the ontology-based specification of `VIARMI`. This description is automatically generated from its annotation interfaces by the analysis using Java Reflection APIs and naming conventions. For example, when the analyzer sees that the annotation interface of `VIARMI` containing the methods `getInspAveragePacketSize()` and `getPrefAveragePacketSizeST()`, it soonly knows three things: 1) The information called "AveragePacketSize" is available for `VIARMI`. 2) The criterion of "AveragePacketSize" is also given. 3) The relationship between the value of 1) and 2) is that `VIARMI` prefers the running conditions that the value of 1) is smaller than the value of 2). It then generates these two terms `InspAveragePacketSize` and `PrefAveragePacketSize` with runtime-investigated values and the relation term `smallerThan`, as shown in Figure 3, when describing a running `VIARMI` component based on our ontology. The detailed algorithm in transforming annotation interfaces to DAML+OIL language is given in Appendix. Figure 3 is the results generated from the Algorithm in the appendix with input given in Figure 2.

3. The CMS architecture

In order to explore the real effects of runtime component compositions on improving the component-based applications, we have designed and partially implemented a system called Component Management Service (CMS) that incorporates the techniques mentioned above to optimize the component configuration of the programs. To subscribe this service, the user program exports the annotated components to CMS and use the returned proxy objects in the following program context. CMS will then continuously analyze the components it holds, evaluate their suitability, search for better replacements, and perform the substitution safely. CMS is with a flavor of JIT compilers, but performs a coarse-grained component replacement for performance optimizations.

The architecture of CMS is shown in Figure 4. We explain each component below.

1. **Dispatcher** is the main block that enables the runtime substitution of components. When the user program registers an object to CMS, the dispatcher keeps it in

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/
                           22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/
                                 rdf-schema#"
  xmlns:daml="http://www.daml.org/2001/03/
                                    daml+oil#"
  xmlns:xsd="http://www.w3.org/2000/10/
                                   XMLSchema#"
  xmlns:cao="http://ariadna.cs.nthu.edu.tw/web/
                                 2003/06/cao#"
  xmlns:caodt="http://ariadna.cs.nthu.edu.tw/web/
                              2003/06/cao-dt#"
>
  <cao:FunctionItf rdf:ID="#RMI">
    <cao:hasImplementation rdf:resource=
                                    "#VIARMI"/>
  </cao:FunctionItf>
  <cao:Implementation rdf:ID="VIARMI">
    <cao:implements rdf:resource="#RMI"/>
    <cao:withAnnotation rdf:resource=
                           "#ReqViplVersion"/>
    <cao:withAnnotation rdf:resource=
                    "#PrefAveragePacketSize"/>
    <cao:withAnnotation rdf:resource=
                     "#PrefAverageBandwidth"/>
    <cao:withAnnotation rdf:resource=
                       "#PrefAverageLatency"/>
    <cao:withAnnotation rdf:resource=
                           "#ExtViplVersion"/>
    <cao:withAnnotation rdf:resource=
                    "#InspAveragePacketSize"/>
    <cao:withAnnotation rdf:resource=
                     "#InspAverageBandwidth"/>
    <cao:withAnnotation rdf:resource=
                       "#InspAverageLatency"/>
  </cao:Implementation>
  <cao:ExternalRef rdf:ID="ExtViplVersion">
    <daml:hasValue>v1.01</daml:hasValue>
  </cao:ExternalRef>
  <cao:Inspection rdf:ID="InspAveragePacketSize">
    <daml:hasValue>467 Byte</daml:hasValue>
  </cao:Inspection>
  <cao:Inspection rdf:ID="InspAverageBandwidth">
    <daml:hasValue>398 Kbps</daml:hasValue>
  </cao:Inspection>
  <cao:Inspection rdf:ID="InspAverageLatency">
    <daml:hasValue>78 us</daml:hasValue>
  </cao:Inspection>
  <cao:Requirement rdf:ID="ReqViplVersion">
    <cao:onFact rdf:resource="ExtViplVersion"/>
    <cao:lowerBound>v1.01</cao:lowerBound>
  </cao:Requirement>
  <cao:Preference rdf:ID="PrefAveragePacketSize">
    <cao:onFact rdf:resource=
                     "InspAveragePacketSize"/>
    <cao:smallerThan>512 Byte</cao:smallerThan>
  </cao:Preference>
  <cao:Preference rdf:ID="PrefAverageBandwidth">
    <cao:onFact rdf:resource=
                      "InspAverageBandwidth"/>
    <cao:biggerThan>1.5 Mbps</cao:biggerThan>
  </cao:Preference>
  <cao:Preference rdf:ID="PrefAverageLatency">
    <cao:onFact rdf:resource=
                        "InspAverageLatency"/>
    <cao:smallerThan>100 us</cao:smallerThan>
  </cao:Preference>
</rdf:RDF>
```

Figure 3. The specification of `VIARMI` in our ontology format.

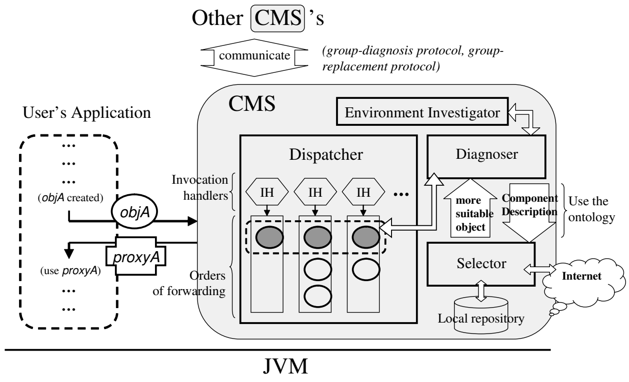

Figure 4. The architecture of CMS.

a new forwarding list and creates a invocation handler associated with the list. The invocation handler contains logics finding the composition points, doing state synchronization, and cooperating with the other invocation handlers, probably in another CMS in the distributed environment for concurrent compositions.

2. **Diagnoser** probes the status of the currently targeted components to judge the suitability of them. It uses Java Reflection APIs to get the annotation interfaces they implement and then extract information through the interfaces. With the help of naming conventions, the diagnoser compares the method names in the `Requirement` and `Preference` interfaces with those in the `Inspection` and `ExternalRef` interfaces, to see if the component is a correct one for use in terms of performances. If so, the diagnoser packs up the information about the wished component based on the component description ontology and ask Selector to search and recruit it. The diagnosis process may involve queries on other CMSs if the component belongs to a cooperation group. The group-diagnosis protocol is then used to discuss for reaching the consensus about the group-level composition.

3. **Selector** is taking requests from the diagnoser and searching for suitable components. It will look into the local repository or the Internet with the help of the component description ontology. A component searching engine may reside on the Internet providing precise searching of components as services. In the CMS that runs on small devices such as PDAs, it's appropriate and reasonable to move this block to a powerful servers for resource saving.

4. **Environment Investigator** implements the `Environment` interface to provide environmental information for the use of the diagnoser. This information will also be packed into the description

Table 1. The conjugate gradient solver computation time (seconds) with and without the proxy overhead on the dense and sparse matrix component.

	Direct Ref.	Dynamic Proxy
DenseMatrix	143.318	143.532
SparseMatrix	6.193	6.213

of the wished component for the selector to make decisions. This is related to state approximations of architectures[8]. Another job of the environment investigator is to notify the diagnoser the changes of the environments such as the plug-and-play hardware devices, the in-ranged wireless services, or newly installed software libraries.

4. Experiment

In the experiments, we use CMS to enable runtime component compositions in three application scenarios. The examined components are annotated by our component specification. The experimental results thus show the potential benefits of applying our component technology. The overhead of runtime component compositions is also revealed in these examinations and turns out to be small in experiments.

4.1 Matrix Manipulation

We use a CFD program as our first application scenario. In the CFD codes for the parallelization of pressure correction method, it will need matrix components for the conjugate gradient solver. Possible implementation of the matrix components are dense matrix components, sparse matrix components, and parallel matrix components. We examine the performance of the conjugate gradient solver that incorporates these components and CMS. The hardware configuration is a 4-node PC-cluster connected with 100Mbps ethernet, and each node is running on 800MHz AMD Athlon CPU with 256MB memory. The software environment includes Sun Java SDK 1.4.1_01, PVM 3.4.4, and jPVM 1.1.4 over Linux kernel version 2.4.8.

In the first set of our experiment, the conjugate gradient solver is executed by using the sequential and dense matrix component, the diagnoser of CMS then tries to evaluate the suitability of the dense matrix component. If the dataset held are mostly sparse, a mismatch of the results of running `getPreferredSparsity()` and `getCurrentSparsity()`, which are in the annotation interfaces that the dense matrix component implements,

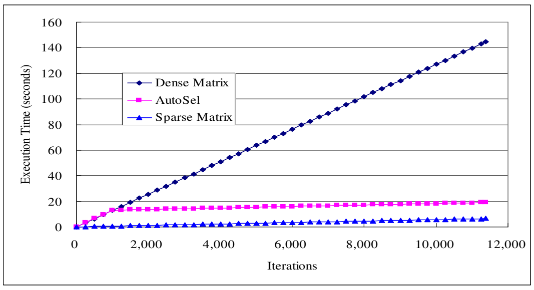

Figure 5. The conjugate gradient solver running time on dense matrix, sparse matrix, and the runtime composition from dense to sparse.

will arise and cause the diagnoser to issue requests to the selector for searching a more suitable one. The selector in this case picks up the sparse matrix component because the returned value of running `getPreferredSparsity()` on it is closer to the current sparsity of data than the dense one. Due to space limitation, we leave details of annotation interfaces and and ontlogy of matrix components in [4].

Table 1 first reveals the potential benefits of using a sparse matrix component instead of a dense matrix component in the CG solver with the sparse dataset in [12, page 352, Fig. 4]. As we implement composition with Java proxy mechanisms, Table 1 also shows the overhead of proxy implementations of matrix components compared to the one without proxy implementations. The overhead of employing proxy mechanisms is very small. As the dataset is actually a sparse dataset in this case, significant performance gain can be expected if the same dataset were executed by a sparse matrix component. We then now incorporate the component composition and CMS system for runtime compositions. Figure 5 then compares the CG solver running time on the dense matrix component, sparse matrix component, and the dense matrix component which changed in the middle of execution to the sparse one, which is denoted by the `AutoSel` line. In Figure 5, the x-axis is the iteration number and the y-axis represents the execution time in seconds. Note that the number of iteration is large due to the requirements of numerical accuracy. The selector of CMS automatically selects the sparse matrix component during the CG solver at early stage of computations after examining sparsity of programs. The runtime composition overhead including proxy, selection, and object serializations and the overhead is amortized. We can observe that the version with the runtime composition gains around 7 times speedup than the version with dense matrix components.

We now look at the aspects of optimizations with parallel environments. We now have the diagnoser to

Table 2. The conjugate gradient solver computation time (seconds) with and without the proxy overhead on parallel matrix.

Procs.	Direct Ref.	Dynamic Proxy
1	150.963	151.650
2	91.011	92.721
4	64.293	64.679

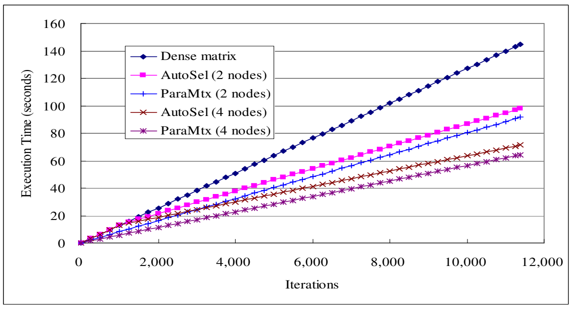

Figure 6. The conjugate gradient solver running time on dense matrix, parallel dense matrix, and the runtime composition from dense to parallel dense.

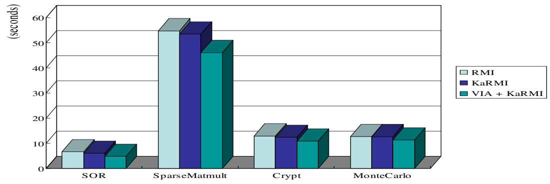

Figure 7. Java Grande Forum MPJ benchmarks with RMI component adaptations.

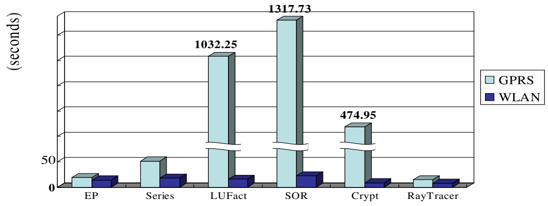

Figure 8. Adapting RMI component over wireless environments

observe if a sequential application is a long running execution. In this case, the selector then picks up the parallel dense matrix component due to the match of the names `getPreferedParallelNode()` in the annotation interface it implements and `getAvailableParallelNode()` in the environment interface `PVM`. Table 2 again reveals the potential benefits of switching to the parallel dense matrix component. The result in Figure 6 shows the composition of parallel dense matrix components (the `AutoSel` lines) does improve the original solver. In Figure 6, the upper line represents the long execution time of the one with sequential matrix component. Next, it shows both 2-node and 4-node execution. Each case is with two lines quite close to each other. One of the two lines in the low parts are the one running initially with parallel configurations. The `AutoSel` one is initially executed by sequential programs, and then the selector composes the program into a parallel matrix version. Object-serializations are needed here as parts of overheads to provide consistency state. We can see the performance benefits out-perform the overheads.

4.2 Cluster Computings with Specialized RMI

The second application scenario is the RMI communications in the cluster computings. The high-performance clusters are usually equipped with special high-speed, low-latency inter-networks. The cluster used in the previous experiments is also equipped with VIA [19] network. Since the traditional intercommunications of parallel programs like barriers, broadcastings, etc. are usually latency-sensitive, we will demonstrate our component technology by adapting RMI components over ethernets to RMI components over VIA for gaining the low-latency benefits from the specialized hardware. In our clustered environments, we support three RMI components including RMI component over TCP/IP and ethernets, KaRMI [16] over TCP/IP, and KaRMI over VIA. The KaRMI is adopted from the work in KaRMI [16] and is designed to be easily ported to different networks. We experiment with compositions of Java RMI components with the Java Grande Forum MPJ Benchmarks [7]. The benchmark suite includes successive over-relaxation (SOR), sparse matrix multiplication (SparseMatmult), IDEA encryption (Crypt), and monte carlo simulation (MonteCarlo). The additional interfaces added for RMI components will allow us to choose the proper components for given environments. In this case, the match of the returned value of `getRequiredVipl()` and `getCurrentVipl()` called on the VIA technology object and the environment investigator will have CMS find RMI version over VIA as a suitable one and compose it in. Figure 7 shows the potential performance benefits in properly adopting the right RMI version suitable for given system environments. The KaRMI over VIA has the best performance if the environment status can be probed, and the annotation interface of RMI can be picked up. Note that our annotation information and composition mechanisms can also be used in a dynamic situation, where some of the network traffics are loaded so that a selection of RMI components can be made based on network environments which are less loaded.

4.3 RMI over Heterogeneous Wireless Networks

Finally, we have the application scenario for RMI components adapting over heterogeneous wireless environments. In our research site, we have previously built up Java RMI over GPRS, WLAN, and bluetooth environments under university research excellence project [13]. The GPRS system for our experiment is equipped with HLR, BSS, SGSN and GGSN. In addition, a HA (Home Agent) providing mobile service for GPRS network is connected to GGSN via Gi interface. For Java applications written with RMI components over heterogeneous wireless environments, the CMS environment can help interact with applications for choosing proper RMI components. For example, an application can be initially running RMI over GPRS environments, but as the mobile device moved into the range of a WLAN, the RMI component over WLAN can be chosen. We demonstrate this idea with experiments. A mobile node is communicating with the PC, and running the MPJ benchmarks through the RMI over GPRS. When the mobile node moves into the range of one WLAN, the diagnoser of CMS can decide to adopt the RMI component over WLAN because it has wider bandwidth. Figure 8 shows the potential benefits. The applications are from Java Grande (MPJ) benchmarks. The one note as GPRS is the application always running RMI component over GPRS. The one note WLAN is the one initially running with GPRS, but then the CMS finds out that the application is in the range of WLAN in the the very early stage of the computation. The RMI component over WLAN is quickly adopted. Significant performance are gained in this case.

5. Related Work

In the works related to parallel component environments for component composition and collaborative frameworks, the autonomic server components [8] are proposed to dealing with modern server designs and to reduce the system-level complexity, as each service is often a composition of many other services. The ideas of the mechanism include an operational definition of an autonomic component on a server, and the key features it must possess to be effective in the emerging environment. An interface model for parallel component architecture [2], which allows reusable component modules to be developed independently of each other, and mediates, monitors, and optimizes interactions between such components. The PSEware project [11] is a prototype component based problem solving environment. A PSE called the Linear System Analyzer (LSA) have built for the manipulation and solution of large sparse linear systems of equations. The GrADS project [1] was developed to provide tools and technologies for the development and execution of applications in a Grid environment. Its goal is to simplify distributed heterogeneous computing in the same way that the World Wide Web simplified information sharing over the Internet. Our work provides specifications and system support for runtime component compositions. It's a part of our research efforts in supporting high-performance object-oriented environments on heterogeneous network environments.

Several projects utilize wrapper or dynamic proxy for component composition and method interception, Dalang [23] is an extension for Java that introduces behavioral runtime reflection. It uses class wrappers to implement a simple metaobject protocol where a metaobject controls the behavior of a base level object. The metaobject can redefine the handling of method calls in order to add desirable properties such as fault tolerance or security. The Aroma system [15], which is Java-based middleware that aims to exploit the Java RMI to replicate objects for availability and adaptability. The dynamic proxy is used as an interceptor to extend the capability of Java RMI. An abstraction for object interaction in a P2P environment, called query/share (QS) [9], is implemented in Java and relies heavily on the concept of dynamic proxy.

Efficient supports for Java remote method invocation have been important topics for investigations, as RMI provides a layer of abstractions for communications. Research results include an open RMI implementation which makes better use of the object-oriented features of Java [18], ARMI [17] and Manta [14] systems to reduce various drawbacks in RMI and to provide new RMI style systems with added functionality, KaRMI [16] is a better way to implement RMI with exploiting Myrinet hardware features to reduce latencies for high performance, our early work on RMI over wireless environments, and the broad range of RMI applications [3].

6. Conclusion

In this work, we gave early experimental results for a challenging problem in providing specifications for components to enable composition optimizations based on runtime compositions and selections. With the arrival of grid-like architectures and autonomous computing system, we think this is going to be a very important area to explore. It will be a even more challenging issue in the next step to demonstrate the applications of these technologies and concepts for large scale of complex component software. Our proposed ontology specifications and software frameworks should provide a solid basis for further explorations.

Appendix: Translate annotation interfaces into DAML+OIL languages

The following procedure analyzes a component C and generates a DAML-based file that describes it at the time T. Initially the set S_1, S_2, and S_3 are empty, and C is running inside an application doing real jobs or inside an instance pool waiting for applications to pick up and use. For conciseness of the description, the following terms are used:

- $head(M)$: Return "Insp", "Ext", "Req", or "Pref" accordingly if the name of M starts with "getInsp", "getExt", "getReq", or "getPref". Otherwise a null string is returned.
- $tail(M)$: Return the tail part of the name of M if it equals "UB", "LB", "CON", "EQ", "BT", "ST", or "APP". Otherwise a null string is returned.
- $body(M)$: Return the part of the name of M that does not belong to $head(M)$ and $tail(M)$.

The main idea of this procedure is using naming conventions to help automatic analysis of annotation semantics. After knowing the interfaces of a given component using reflection APIs (Step 2), it differentiates the functional interfaces and annotation interfaces by finding out if it is an extension of the interface `Annotation` (Step 3). All annotation interfaces are mandated to extend `Annotation`. After generating the static information about the component (Step1, Step 3, and Step 4), it starts to parse the methods inside the annotation interfaces for annotation semantics.

First, any method in the annotation interfaces named with "getInsp" and "getExt" started is processed (Step 5). When there is a method named "getInspX" presented, it means the component writer has provided the user a way to retrieve the information which is identified as X. The procedure then executes this method on the component to get this information and write it down in the description file. Similarly, those methods named "getExtX" stands for the information X but is only retrieved from outside. X is the identity used to query the environment. A better way is to setup an environmental component that has corresponding "getInspX" method to provide such information.

Second, those methods named with "getReq" and "getPref" started are processed (Step 6). They are named with this form "getYXZ". If the "Y" part of one of these methods equals "Req", its "Z" part will be "LB", "UB", "EQ", or "CON". The annotation semantics represented by this method is that the component "requires" the returned value of calling this method on this component to "be the lower bound of", "be the upper bound of", "be equal to", or "contains" the runtime value returned by "getInspX" or "getExtX" methods. The value of this mehtod will be written down into the description file and the previously identified `Inspection` and `ExternalRef` methods also will be looked up in order to state the relationship with them in the file. Similarly, if the "Y" part equals to "Pred" and the "Z" part equals to "BT", "ST", or "APP", it means there is a preference existing that is "bigger than", "smaller than", or "approximate to". Figure 3 in Section 2.2 is the result generated by this Algorithm with the input given in Figure 2 of Section 2.1.

Algorithm 1 Translate annotation interfaces of components into DAML+OIL languages
Begin
Step 1.
Generate header information. Declare the namespaces used. Beside the standard namespaces such as `rdf`, `rdfs`, `daml`, and `xsd`, the `cao` and `caodt` is addressed here to locate our ontology definition. Additional domain-specific ontologies can also be refered here.
Step 2.
Use Java reflection APIs to find the interfaces that C implements. Put them into S_1.
Step 3.
FOR any interface I in S_1
  IF I doesn't extend the interface `Annotation`
    Move I into S_2
    Declare I as a `FunctionItf` *class*
    Declare a `hasImplementation` *property*
      from I to C
  END
END
Step 4.
Declare C as a `Implementation` *class*
FOR any interface I in S_2
  Declare a `implements` *property* from C to I
END
FOR any interface I in S_1
  Declare a `withAnnotation` *property*
    from C to I
END
Step 5.
FOR any interface I in S_1
  FOR any method M defined in I
    IF $head(M)$ equals "`Insp`"
      Move M into S_3
      Declare M as a `Inspection` *class*
      Call M on C and get the returned value V_T

Declare a `hasValue` *property* from M to V_T
ELSE-IF $head(M)$ equals "`Ext`"
Move M into S_3
Declare M as a `ExternalRef` *class*
Fetch the value V'_T from the external environment using the identity $body(M)$
Declare a `hasValue` *property* from M to V'_T
END
END
END

Step 6.
FOR any interface I in S_1
FOR any method M defined in I
IF $head(M)$ equals "Req"
Declare M as a `Requirement` *class*
Call M on C and get the returned value V_T
Declare a `lowerBound`, `upperBound`, `withValue`, or `contains` *property* from M to V_T accordingly if $tail(M)$ equals "LB", "UB", "EQ", or "CON"
FOR any method M' in S_3
IF $body(M)$ equals $body(M')$
Declare a `onFact` *property* from M to M'
END
END
ELSE-IF $head(M)$ equals "Pref"
Declare M as a `Preference` *class*
Call M on C and get the returned value V_T
Declare a `biggerThan`, `smallerThan`, or `approximating` *property* from M to V_T accordingly if $tail(M)$ equals "BT", "ST", or "APP"
FOR any method M' in S_3
IF $body(M)$ equals $body(M')$
Declare a `onFact` *property* from M to M'
END
END
END
END
END

End

References

[1] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, J. Mellor-Crummey, D. Reed, L. Torczon, and R. Wolski. The GrADS Project: Software Support for High-Level Grid Application Development. *International Journal of High Performance Applications and Supercomputing*, Vol. 15, number 4 (Winter 2001): 327–344.

[2] Milind Bhandarkar and L. V. Kale. An Interface Model for Parallel Components. In *Proceedings of LCPC*, August 2001.

[3] Fabian Breg, Shridhar Diwan, Juan Villacis, Jayashree Balasubramanian, Esra Akman, and Dennis Gannon. Java RMI Performance and Object Model Interoperability: Experiments with Java/HPC++. *Concurrency: Practice and Experience*, 10(11–13):941–956, September–November 1998.

[4] Cheng-Wei Chen, The Paradigms of Component Adaptations on Distributed Computing Environments, *Ph.D. Dissertation*, Department of Computer Science, National Tsing-Hua Univ., Oct. 2002.

[5] Chung-Kai Chen. The Component Annotation Ontology, http://pllab.cs.nthu.edu.tw/cao.daml.

[6] Brad Cox. *Object-Oriented Programming: An Evolutionary Approach*, Addison-Wesley, Reading, MA 1986.

[7] Edinburgh Parallel Computing Centre. The Java Grande Forum MPJ Benchmarks.
http://www.epcc.ed.ac.uk/javagrande/mpj.html.

[8] Kattamuri Ekanadham, Joefon Jann, Pratap Pattnaik, Ramanjaneya Sarma Burugula, Donna Dillenberger. Anatomy of Autonomic Server Components. Research Report, RC22637 (W0211-096), IBM Research Division, Thomas J. Watson Research Center, 2002.

[9] Patrick Eugster and Sébastien Baehni Abstracting Remote Object Interaction in a Peer-2-Peer Environment. In *Proceedings of Joint ACM Java Grande - ISCOPE 2002 Conference*, pp. 46–55, June 2002.

[10] I. Foster, C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit, *Intl J. Supercomputer Applications*, 11(2):115–128, 1997.

[11] D. Gannon, R. Bramley, T. Stuckey, J. Villacis, J. Balasubramanian, E. Akman, F. Breg, S. Diwan, and M. Govindaraju. Component Architectures for Distributed Scientific Problem Solving, *Special Issue of IEEE CS&E Magazine on Languages for Computational Science and Engineering*, April–June 1998.

[12] John R. Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333–356, January 1992.

[13] Hong-Wei Lin, Jyh-Cheng Chen, Ming-Chia Jiang, Ching-Yang Huang. Integration of GPRS and Wireless LANs with Multimedia Applications. *Proceedings of IEEE Pacific Rim Conference on Multimedia*, 704–711, 2002.

[14] J. Maassen, R. van Nieuwport, R. Veldema, H. E. Bal, and A. Plaat. An efficient implementation of Java remote method invocation. *ACM PPoPP*, pages 173–182, Atlanta, GA, May 1999.

[15] N. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Interception in the Aroma System. *Proceedings of the ACM 2000 Java Grande Conference*, San Francisco, CA (June 3–4, 2000), pp 107–115

[16] C. Nester, M. Philippsen, and B. Haumacher. A more efficient RMI for Java. *Proceedings of the Java Grande*, pages 152159, 1999.

[17] R. Raje, J. Williams, and M. Boyles. An asynchronous remote method invocation mechanism for Java, *in CPE*, Vol. 9(11), 1207–1211, 1997.

[18] G. K. Thiruvathukal, L. S. Thomas, and A. T. Korczynski. Reflective Remote Method Invocation. In *Concurrency: Practice and Experience*, 10(11–13):911–925, September–November 1998.

[19] UC Berkeley Millennium. *Virtual Interface Architecture*.
http://www.cs.berkeley.edu/ philipb/via/.

[20] Frank van Harmelen, Peter F. Patel-Schneider and Ian Horrocks. Reference description of the daml+oil (march2001), *Ontology markup language*, March 2001,
http://www.daml.org/2001/03/reference.html.

[21] W3C. *Extensible Markup Language (XML)*, 2001. http://www.w3.org/XML/.

[22] W3C. *Semantic Web Activity: Resource Description Framework (RDF)*, 2001. http://www.w3.org/RDF/.

[23] Ian Welch and Robert J. Stroud. From Dalang to Kava - The Evolution of a Reflective Java Extension. *Reflection* 1999: 2–21