published as: W. Huang, U. Roth, Christoph Meinel: W. Huang, U. Roth, Christoph Meinel; Proceedings of International Workshop on the move to meaningful Internet systems (WRSM 2003); WRSM 2003 Sicily (Italy), 2003, pp. 950-959; isbn: 3-540-20494-6.

A Flexible Middleware Platform with Piped Workflow

Wanjun Huang, Uwe Roth, and Christoph Meinel

Department of Computer Science, University of Trier
D-54286 Trier, Germany
{huang,roth,meinel}@ti.uni-trier.de

Abstract. Middleware emerges as an excellent solution for the complicated distributed computing application. But as the appearance of new devices and new applications, the inflexibility of traditional middleware system becomes more serious and urgent. In this paper we propose a new flexible middleware platform which adopts the technologies of piped workflow and computational modules to provide a modular and extensible platform for future applications. The piped workflow provides a very flexible mechanism to organize all computational modules working together. During the running time, all computational modules communicate only with data channel of the piped workflow through which they can keep extremely independent, and the flexible flow control strategy makes the application programmer convenient to arrange all functional components for variable customer requirements.

1 Introduction

The emergence of middleware has solved a serial of problems arose by applications widely distributed on network, and evolves the distributed computing model from client-server into three tiers architecture. As the middle tier component between the low level operation system and the top application software, middleware helps programmer easily and quickly build distributed business application without considering some common but complicated problems, such as heterogeneity of operation system, complexity of communication, concurrent interoperability, system stability, transfer security and so on. Almost all traditional middleware solutions adopt the mechanism of block box and have gained big success in providing remote procedure access. However, as appearance of new devices and new applications, the inflexibility and limitations of traditional middleware become more serious and absorb many researchers' interests and concerns. When network just comes into our life, how to easily build a distributed application is the first task. But now, as network become more pervasive, and as portable handheld devices become more popular, the more important problems become how to make the middleware more extensible, reusable and adaptive for unknown future applications and how to build middleware itself more easily. In order to provide more convenience for the application writer, the middleware has to undertake more functionalities and responsibility, which make it hard to

R. Meersman and Z. Tari (Eds.): OTM Workshops 2003, LNCS 2889, pp. 950–959, 2003.
© Springer-Verlag Berlin Heidelberg 2003

complete the implementation of a fat middleware server. Additionally, modularity and extensibility are also another two big unsolved problems. The traditional middleware are composed of fixed components with fixed policies that can't meet the diversity needs of varying applications environment. Although some solutions [4], [5], [8] have been proposed to dynamically customize the middleware components to adapt the variety of client environments, these varieties can be predicted and have been considered during the design of original architecture. For every unknown or unpredictable application, some original source codes have to been modified to allow the new processing components working well along with old ones. For example, consider an e-business application running on a distributed environment. Initially the transactions processing are some common remote data access and remote procedure invocation, so the traditional middleware solution is capable to deal with them. Subsequently, the middleware server is required to provide multimedia services, so stream transfer protocol, MPEG encode/decode and quality of service have to been added into inner key components. Eventually, company's business services want to be expanded to mobile commerce, so some filter components are also needed for mobile computing. If all these changes will inevitably arise to the modification of source code or even the structure of system, then each update of middleware will bring big price of time consuming and hard work. So it's necessary to explore and research more flexible middleware architecture to improve the efficiency of system implementation and make the key components of middleware more independent to adapt for the future update.

Component oriented software development, such as Java Bean and ActiveX Control, support the construction of sophisticated system by assembling a collection of components software with the help of visual tool or programmatic interfaces. However, they have to integrate in the level of source code and support little for dependences management, which are important for component to be freely loaded into or unloaded from inner workflow to work fluently in distributed system. Here we propose a new flexible middleware platform, which adopt a piped workflow and computational module to achieve the integration of independence and the flexibility for future extension. In the following, firstly we will introduce the architecture of new middleware platform and explain some important components. Subsequently, the computational module and piped workflow will be described in details. And then we discuss the related work recently proposed by other researcher. At last conclusion will be made to summarize its features and outline our future work.

2 Architecture

To testify the new ideal, we have designed and implemented middleware platform - Smart Data Server V2.0 (SDS 2), which has adopted the technologies of computa-tional modules and piped workflow. All components can fall into three catalogues: infrastructure, computational modules and application services. The infrastructure is laid on the bottom level that contains piped workflow and some core services. Computational modules live only inside of piped workflow and are

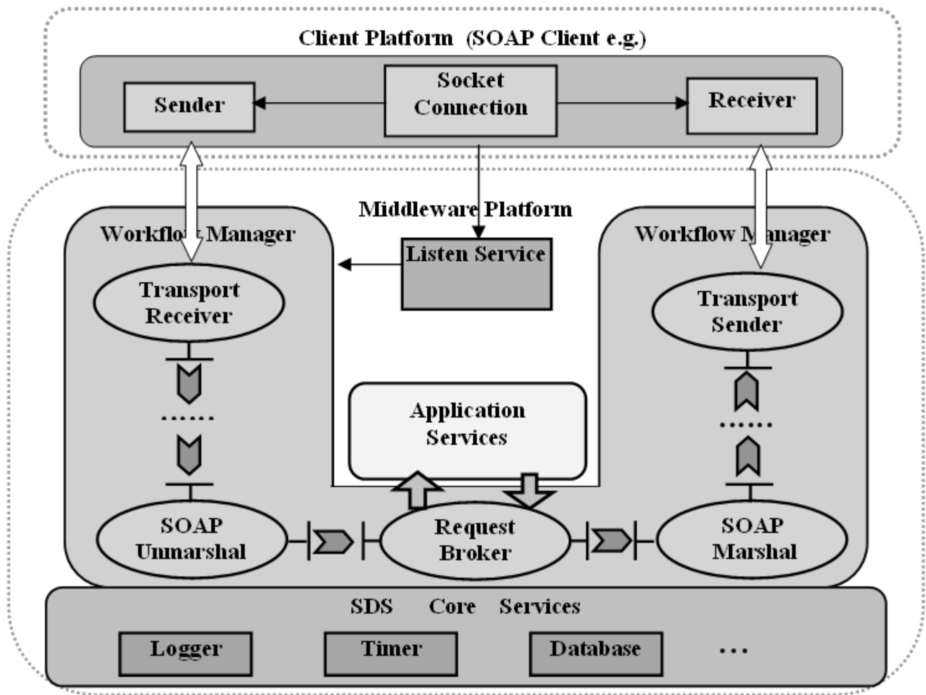

Fig. 1. Piped workflow middleware architecture

the primary components to process client request. The relation diagram of all these components is depicted as figure 1.

2.1 Core Services

The Core Services provides some fundamental services that will be used in the whole middleware platform. Not only in the workflow manager and computational modules, these services can also be used directly in the application services. SDS 2 Core Services include Logger, Timer and Database etc which provide consistent services using the technology of thread instead of Application Programming Interface (API). Logger Service is used to record error, warning, debug and running state information which are convenient for programmer to debug the system and for administrator to check the middleware server. Timer service can activate some task at one specific time or periodically that is very useful for the some special application, such as checking email etc. Listen service is also an important part of Core Services, and its responsibility are to listen socket request and establish socket connection with client that will be transferred to workflow manager to activate a workflow processing routine.

2.2 Workflow Manager and Modules

Workflow Manager provides the service for piped workflow to organize all relevant modules working together. Here the implementations of computational modules contain TransportReceiver, SOAPUnmarshal, RequestBroker, SOAP-Marshal and Trans-portSender. In our implementation, we also finish another two modules, namely IPTPUnmarshal and IPTPMarshal, to transfer the request and response messages using self-defining protocol. - Information Package Transfer Protocol[1]. So, two transfer protocols are available at the same time according different flow solutions.

2.3 Application Service and Client Tools

Application service is an application repository where store the current services provided for client. These services can be accessed directly via Java reflection technology. So there is no requirement for user to declare the interface of deployed services, but the price for this flexibility is the limitation of usage of Java Reflection. The client tool depends on the transfer protocol. One of our adopted protocols is Simple Object Access Protocol (SOAP). It's an international standard protocol, so any client package developed by companies or open organizations can be used as our client tools, such as Apache SOAP client package, Microsoft SOAP toolkit client package etc.

3 Computational Module

Component technology has showed its power in the development of sophisticated software system, such as Java Bean and ActiveX Control. In the middleware system, component should not only keep its characteristics of inner integration and independence, but also hold a high ability to be flexibly controlled and integrated into the existed system. At present component technology has already been applied into most of current middleware systems or products, but different components are always mixed into each other in some extent when they work together. Here the proposed components structure calls computational modules that can avoid these confused problems. As described in figure 2, computational module exhibits its outer behaviours in the form of following four items:

- *Input Interface* that represents what computational module need for internal processing.
- *Output Interface* that shows the processing result of module.
- *Properties* that describe initial setting for module.
- *Requirements* that list the basic conditions of module that are used to verify inputted data.

Module exhibits its IO behaviors through input and output interfaces and hides all the inner implementation detail as a black box for application writer. To reify the properties, module contain a serial of basic methods, such as methods

for initialization and release, to ensure it can be load and unload in any time. Requirements are set to control whether inputted data are qualified. This step is benefit to optimize processing, and also very crucial for workflow manager to decide which is the next module after current one. The construction of computational module is based on the technology of thread, which make it able to run independently. During the reification of input and output interface, all modules have to implement a behavior interface, which inherit from a common IO interface to generate the common IO behaviors. Workflow manager can access all IO behaviors of every module via the way of looking up member methods of its behavior interface. Input interface tells workflow manager what it needs, and output interface tells what it will produce. All the methods of input interface will be executed by piped workflow to get the required data from data channel before the computational module runs, and it will also produce its results into data channel after finishing of running. So speaking strictly, all modules keep no contact with each other and they just communicate with data channel of workflow manager. This independent property of computational module brings much flexibility to add a new one into the system.

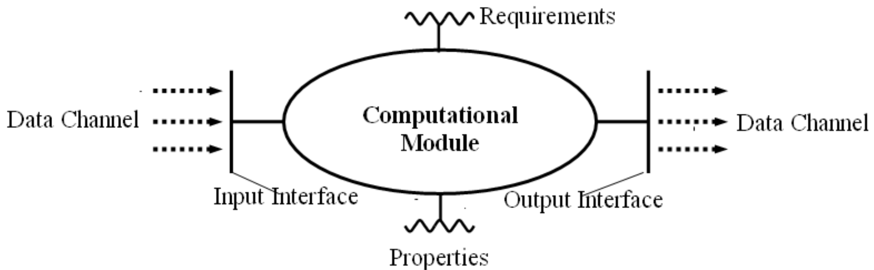

Fig. 2. Structure of computational module

During the procedure of implementation, programmer can develop many modules according the actual requirements. But not every module has to be engaged in the processing of one request, and it depends on the workflow solutions and client request to decide which module will be used for one processing. In future more modules can also be freely added here to adapt the middleware for new application, such as MEPG encode and decode modules for multimedia application etc.

4 Piped Workflow

In traditional middleware systems inner workflow is always integrated closely with processing components which result in the inextensibility for future extension. To achieve more flexibility for future components and unknown requirement, we propose a piped workflow to adapt this variation. The piped workflow consists of Workflow Nodes, a Data Channel and a Flow Control.

4.1 Workflow Nodes

There are three kinds of workflow nodes: Program-Nodes, Flow-Nodes and End-Nodes which are all inherited from a same parent node. Program-Node is a special node used in workflow program. It is not only tied with a start module where the flow control starts, but also used to control other modules and specify sub-routines. Flow-Node is the core node inside of workflow. It is tied with most computational modules, which take charge of the primary work of request processing. End-Node is tied with last processing module. In fact, all these three kinds of nodes will not do any real work for request processing, and they are just used to control the workflow whose primary task is to decide which module is the next one. The actual works are done by computational modules that are associated with workflow nodes through the implementation of pipe interfaces, as depicted in figure 3. Using the corresponding relation between computational modules and workflow nodes, piped workflow mechanism can flexibly control the computational modules through the control of workflow nodes. When a new module is added, what the administrator needs to do is to bind the new module with a specific workflow node in the configuration file.

4.2 Data Channel

The traditional middleware components can not keep absolutely independent because they have to directly communicate and exchange data each other. In piped workflow a special media - data channel is provided to exchange data among modules. Here there are two ways to realize its functionality of data exchange. One is data pool, and another is pipe, as described in figure 4.

Data Pool. Data pool is similar with the conception of bus in area of Integrated Circuit (IC). It is a common media where the communications between

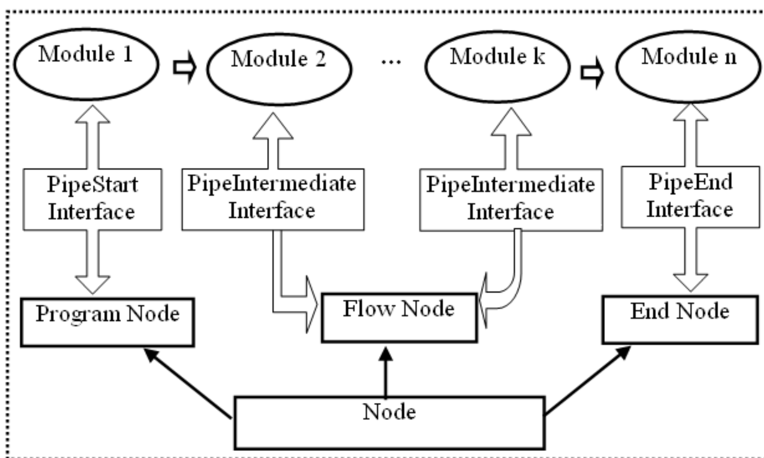

Fig. 3. Relation between workflow nodes and computational modules

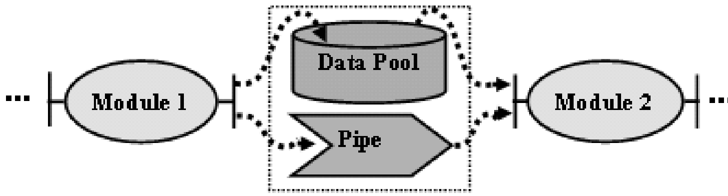

Fig. 4. Data channel

different components are avail-able. Data pool provides services for all relevant modules during one processing procedure. It will be generated when a processing of one request starts and destroyed when the processing of request finishes. The implementation of data pool contains a public temporary storage and communication ports between data pool and modules. Each module has to define which data it needs from data pool and which data will be produced to data pool. These functionalities are realized through the implementation of an IO behavior interface extended from a common interface. In each I/O-behavior interface the set- and get-methods are defined to represent the input and output behaviors respectively. Under the management of workflow manager these variable data are exchanged between the modules and data pool through Java reflection technology. Each set-method of the module will be checked for its associated input data after the module is initialized and each get-method will also be checked to get result after execution of the module. If later the module is replaced by another different module, the interface behavior of new module should not be changed. This strategy ensures that other modules still can communicate with such an old module whose entity has been changed.

Pipe. Data pool act as the role of software buses where multi-modules can take the same output of another module as their common source data. It is very convenient for data communication, but can not offer any dependencies information between adjacent modules. So pipe is proposed to link relevant modules and reveal the dependences of them. Through the way of pipe control messages can be transferred to manage the operation and dependency of each module. Pipe consists of several independent sub-pipes that serve only for two adjacent modules. Each sub-pipe contains a pair of ObjectIO that includes ObjectIn and ObjectOut. ObjectIn is used for previous module to transfer data from module to pipe, and ObjectOut is employed for latter module to transfer data from pipe to module. In one pair of ObjectIO, ObjectIn and ObjectOut share the same temporary memory, so the dependency messages can be transferred from the first module to last one via a serial of sub-pipes. Additionally, different type of computational module contains different composition of ObjectIO. For example, a modules implementing PipeIntemedia Interface contains both ObjectIn and ObjectOut. But the computational module implementing PipeStart Interface or PipeEnd Interface only contains ObjectOut or ObjectIn respectively. Using such a pipe mechanism, the dependencies of different modules are represented and control messages can be transferred automatically and orderly.

4.3 Flow Control

The flow control is a control strategy for all nodes of the workflow program. It decides which node will be processed after the current one. Here two factors result in the flexibility of flow control. One is the structure of computational module and data pool that can arrange the node freely according requirements. Another factor is the dynamical control strategy. It means the result of the flow control is not a fixed link of all nodes and will vary dynamically depending on the flow solution and the execution state of each node.

To manage the workflow nodes, different flow solutions should be configured in the configuration file in advance. Just as figure 5, two flow solutions are illustrated where one solution comprises nodes linked by red arrowhead and another one is represented through nodes linked by grey arrowhead. In a valid solution there should be only one module set to start-node from which the processing work start and at least one module directly head to end-node. For a selected flow solution, such as one indicated by red arrowhead, it may also have different routes during the running time. The real arrowheads indicate the normal default route, and dashed arrowheads indicate other possible routes. When a module is running, it will first check the requirements according input-ted data, and then return an execution state with which workflow manager can decide next route. For example, in the red arrowhead solution of figure 5, the node "A" will go to node "B" if it returns a normal execution state code "1", and it will go to node "D" if the returned execution state code is "2". Because computational module runs as an independent thread, acquirement of execution state does not mean the end of module execution. The execution state is just to indicate the processing response of current module for next module according the inputted data and requirement of current module.

5 Related Work

Component oriented software development has become the popular accepted tendency for reusable and extensible distributed application. But how to represent the dependencies of different components and how to organize these components

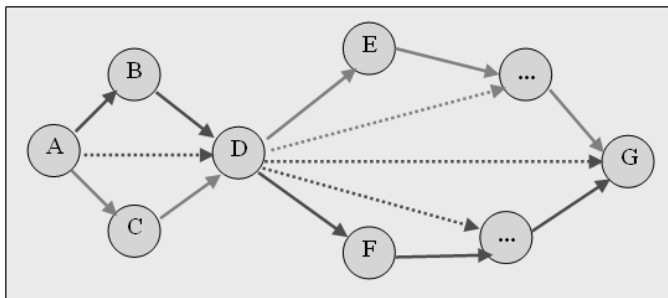

Fig. 5. Paradigm of flow chart

still remain many problems. In [6], authors proposed two distinct kinds of dependences to manage the dependences of components in distributed system. One is prerequisite that are the requirements for loading an inert component into the running system. Another is dynamic dependency that is used to manage the loaded components in a running system. To reify the dynamic dependencies, Kon et al. design a Component Configurator that is responsible for storing the runtime dependencies between a specific component and application components and other system. Each component C has a set of hooks to which other components can attach. These hooked components are the components on which C depends. There might be some other components (called clients) that depend on C. Through the communication and event contact between each component with hooked components and its client, dynamic reconfiguration is enabled for components that are already running. But Component Configurator just define the dependences of component, it has no consideration of the communication between different components. So the component has also to define the same specific behavior interfaces with that of alternative components, just like traditional way. This limitation also restricts the flexibility degree of dynamic reconfiguration. G.S.Blair et al. [3], [4] proposed a configurable and open middleware platform based on the concept of reflection. In their reflective middleware, they introduced open binding and component framework to support the construction of meta-space. Component framework consists of primitive components, which include different kinds of basic and indivisible functional unit, and composite components that represent the configuration of primitive components and composite components. Inside of open binding the communication of different components is realized through the implementation of local binding that is pity to have not been clarified the implementation details. The rather similar idea can be found in [10], Shrivastava present a workflow based model for distributed applications. In their model, workflow schema is used to represent the structure of tasks in a distributed application with respect to task composition and inter-task dependencies. Task controller is used to guide the workflow execution, just like our workflow nodes. But they have not explained clearly and in detailed the conception of computational module and data channel.

6 Conclusion and Future Work

Middleware masks the problem of building distributed application among heterogeneous environment. But the complexity of distributed network and new coming requirement make the construction of middleware more difficult that bring an urgent requirement for high reusability, modularity and extensibility. In this paper we present technologies of piped workflow and computational modules to provide an extensible, configurable and flexible mechanism for the construction of middleware architecture. The computational module exhibits explicitly the properties, requirements and its IO behaviors. Under the management of piped workflow communication between different modules are available and can keep absolutely independent in the level of source code. Additionally, piped workflow

provides an extreme flexible mechanism to schedule all the modules working together according a variable flow control strategy. But in our system flexibility is still not enough. It just provides the ability to arrange modules freely in the server side, and client has no ability to inspect and modify the components structure of middleware server. So how to improve our middleware to gain more ability of reflection is our next research aim. Security is always a hot topic for the field of distributed computing application. Especially when we investigate to provide some inspection and adaptation abilities for client environment, the secure transport, authentication and privilege management have become more important and indispensable to manage middleware platform to meet the need of some crucial and sensitive distributed applications. Additionally, we want to provide some specific services to extend our middleware platform for some specific application area, such as multimedia. But there are still many problems for multimedia supporting, such as stream transfer, quality of service, synchronization and so on.

References

1. U.Roth, E.G.Haffner, T.Engel, Ch.Meinel: The Smart Data Server - A New Kind of Middle Tier. Proceesing of the IASTED International Conference on Internet and Multimedia Systems and Applications, 1999, pp. 362–365.
2. Joo C. Seco and Lus Caires: A Basic Model of Typed Components, Proc. European Confer-ence on Object-Oriented Programming, Cannes, France 2000.
3. Nikos Parlavantzas, Geoff Coulson, and Gordon Blair: Applying Component Frameworks to Develop Flexible Middleware. Workshop on Reflective Middleware, April 7–8, 2000, New York, USA.
4. G. S.Blair, G. Coulson, P. Robin, and M. Papathomas: An Architecture for Next Generation Middleware. In Proceedings of Middleware'98, pages 191–206. Springer-Verlag, Sept. 1998.
5. Fabio Kon, Manuel Roman, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhaes, and Roy H. Campbell: Monitoring, Security, and Dynamic Configuration with the dy-namic TAO Reflevtive ORB. IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000). New York. April 3–7, 2000.
6. Fabio Kon and Roy H. Campbell: Dependence Management in Component-Based Distributed Systems, IEEE Concurrency, 2000. 8(1): p. 26–36.
7. D.C. Schmidt and C. Cleeland: Applying Patterns to Develop Extensible ORB Middleware, IEEE Comm. Magazine, IEEE CS Press. Los Alamitos, Calif., vol. 37, no. 4, 1999, pp. 54–63.
8. Mark Astley, Daniel C. Sturman and Gul A.Agha: Customizable middleware for modular distributed software. Communications of the ACM, v.44 n.5, p. 99–107, May 2001.
9. Mark Astley, Gul Agha: Modular Construction and Composition of Distributed Software Architectures. PDSE 1998: 2–12.
10. S.K. Shrivastava and S.M. Wheater: Architectural Support for Dynamic Reconfiguration of Large Scale Distributed Applications. The 4th International Conference on Configurable Distributed Systems (CDS'98), Annapolis, Maryland, USA, May 4–6 1998.