A Context-Aware Reflective Middleware Framework for Distributed Real-time and Embedded Systems

Shengpu Liu
Lehigh University
19 Memorial Drive West
Bethlehem PA 18015
610-758-4801
shl204@lehigh.edu

Liang Cheng
Lehigh University
19 Memorial Drive West
Bethlehem PA 18015
610-758-4801
cheng@cse.lehigh.edu

ABSTRACT
Context-Aware Reflective Middleware (CARM), which supports application reconfiguration, has been an appealing technique for building Distributed Real-time and Embedded (DRE) systems as it can adapt their behaviors to changing environments at run time. However, existing CARM frameworks impose dependence restrictions and reconfiguration overhead, which makes the reconfiguration time of these frameworks too long (normally in the range of seconds or more) to satisfy the stringent real-time requirements of DRE systems. To improve the reconfiguration efficiency for supporting DRE systems, we have designed a new CARM framework – MARCHES, which offers an original structure of multiple component chains to reduce local behavior change time and a novel synchronization protocol using active messages to reduce distributed behavior synchronization time. MARCHES uses a layered architecture and provides both component-level and system-level reflection to incorporate standard components, a hierarchical event notification model to evaluate contexts, and a lightweight XML-based script language to describe and manage adaptation policies. The MARCHES framework and supported applications have been implemented on PC and PDA platforms. Based on a novel theoretical model, we have analyzed the reconfiguration efficiency of MARCHES and compared it with those of peer CARM frameworks: MobiPADS and CARISMA. Quantitative empirical results show that the reconfiguration time of MARCHES is reduced from seconds to hundreds of microseconds. Evaluations demonstrate that MARCHES is robust, scalable and generates a small memory footprint, which makes it suitable for supporting DRE systems.

Keywords
Reflective middleware; distributed real-time and embedded systems; reconfiguration; synchronization.

1. INTRODUCTION
Distributed real-time and embedded (DRE) systems [1], such as smart grid, aircraft mission planning systems in battlefield, rapid response systems, and vehicle safety systems in unmanned intelligent vehicles, provide an important approach to bridging the gap between the cyber world and the physical world. Generally, DRE systems are large-scale, integrated, and time-sensitive and operate in dynamic and resource limited environments. This challenges system designers and developers when such DRE systems must be developed from scratch. Fortunately, middleware techniques may be used to address this challenge, such as COTS (commercial-off-the-shelf) middleware [2] for reducing application development and maintenance costs, ORB based middleware [3] for enabling component-based system integration, and real-time and lightweight middleware [4] for supporting time-sensitive and resource-limited application.

Recently, adaptive and reflective middleware [5][6] emerges as a promising technique for DRE systems in dynamic environments. It has the ability to inspect its internal states by providing a representation of its internals through a process called reification, and allows the internals to be dynamically manipulated and reconfigured through a process called absorption [10][31]. Furthermore, *context-aware reflective middleware* (CARM) [7][8] can monitor and evaluate real-time situational contexts and reconfigure DRE systems at runtime based on the reflection model. Thus, the systems can adapt autonomously to changing contexts to ensure required quality of service (QoS).

The reconfiguration time of the existing CARM frameworks [9][10] is too long to be acceptable for time-critical DRE systems. The reconfiguration time is normally in the range of seconds or more according to the data reported in literature, but a DRE system requires the total processing time within 10*ms* for time-critical missions [4]. The reconfiguration process of a DRE application consists of two steps: local behavior change, which modifies the structure of the local functional path (or component chain), and distributed behavior synchronization, which coordinates distributed behaviors after the local behavior is changed. For example, in a distributed mobile video transmission application, changing or adding a compression component in a sender program (a local behavior) requires a corresponding change or insertion of a decompression component in the receiver programs (a distributed behavior). The long reconfiguration time of existing CARM techniques is caused by the inefficiency of their synchronization protocols, which are synchronous and require the synchronization participants to be blocked until the reconfiguration process is completed.

In this paper, we propose MARCHES (Middleware for Adaptive Robust Collaborations across Heterogeneous Environments and Systems), which is a CARM framework for engineering DRE systems in dynamic environments. MARCHES uses a layered architecture to monitor contexts and adapt DRE systems to the contexts according to user-defined policies. It supports both component-level reflection for the accommodation of standard components and system-level reflection for the reconfiguration of component connections, contains a hierarchical event notification model to efficiently evaluate comprehensive contexts, and provides a lightweight XML-based script language to describe and manage adaptation policies.

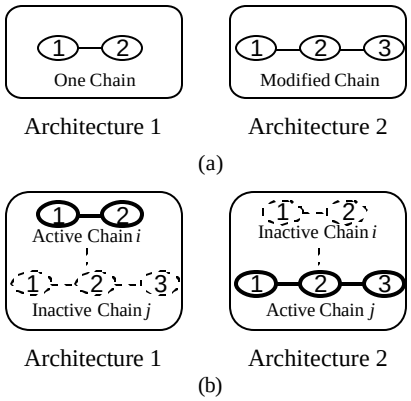


Fig. 1. Dynamic reconfiguration architecture: (a) single component-chain architecture in existing middleware, (b) multiple-component-chain architecture in MARCHES.

1.1 Contributions and Significance

The major contribution of MARCHES is that it has solved the critical issue of the long reconfiguration time of context-aware reflective middleware. Compared to the traditional middleware that supports single component-chain based application architecture (Fig. 1a), MARCHES maintains multiple component chains (Fig. 1b). Therefore, there is a new method proposed for the local behavior change that switches active and inactive chains, which replaces the traditional method of modifying the single-chain structure to reduce the *local behavior change* time.

Further, based on the multi-component chain architecture, an efficient active-message based synchronization protocol is designed to *asynchronously* coordinate the behaviors of distributed programs. The key idea of the protocol is that each application-layer data packet takes an active-message header that indexes the correct component chain, by which the packet receiver processes the data payload. Therefore, the *distributed behavior synchronization time* is also dramatically reduced by eliminating the operation suspension time and buffer clearance time in the existing middleware architecture. The costs introduced by this improvement, such as extra resource consumption and active-message overhead, are extremely low compared to the capacity of the various computing platforms, including mobile devices, as validated by our experiments.

In this research, we have established, to the best of our knowledge, the first generic analytical model for fair comparisons of the reconfiguration efficiency of various CARM frameworks. Using the model analysis and empirical measurement results, we conclude that the reconfiguration time in existing adaptive and reflective middleware has been reduced by several magnitudes, from seconds to milli-seconds.

Real-time is one of the most critical requirements of DRE systems. Although context-aware reflective middleware provides a powerful tool to build adaptive DRE systems, it introduces reconfiguration overhead, which harms system responsiveness and QoS. For example, future unmanned intelligent vehicles with DRE systems can reconfigure their behaviors (direction and speed) to adapt to situational contexts collected at runtime through temporally built ad-hoc and dynamic networks based on vehicle-to-vehicle and vehicle-to-roadside communications. However, the long reconfiguration time may result in critical accidents and loss of lives and property. In fact, two cars could hit each other in 1.5 seconds when they drive face to face based on the 3 second safe distance rule, which requires a vehicle safety system to respond in hundreds of milliseconds. Thanks to the magnitude reduction of application reconfiguration time enabled by the MARCHES framework, a richer set of DRE systems for cyber-physical interactions can be designed and implemented.

1.2 Terminologies

The following terminologies will be used in this paper:

- *Synchronization* is the process of coordinating the behaviors of collaborative programs in a DRE system. When the behavior of a local program is reconfigured to adapt to changing contexts, it requires its peer programs to change their behaviors correspondingly for system consistency.
- *Synchronous synchronization* means that the synchronization is realized through a synchronous method that requires all synchronization participants to complete their behavior changes at the same time and suspend their application-layer operations during this process.
- *Asynchronous synchronization* means that the synchronization is realized through an asynchronous method, in which the local program can resume its operation right after its own behavior is changed for adaptation and other synchronization participants reactively change their behaviors only when they communicate with this local program.
- *Sensor* is the hierarchical context event detector that can organize and evaluate specified contexts at runtime and notify subscribed actuators for adaptation.
- *Actuator* is a reflective component that contains a set of functional components and a meta-interface. The functional components form a functional path or component chain, which process application-layer data. The meta-interface can represent its internal states and reconfigure the actuator behaviors at runtime though component parameter tuning and chain structure reconfiguration.
- *Active actuator* means that the actuator status is active. There is one and only one actuator active at any time and only the component chain in the active actuator processes application-layer data. Various actuators can be activated or deactivated to adapt to changing contexts according to user-defined policies.
- *Proactive actuators* are the actuators constructed at the system initialization phase to process local data. They can proactively change their behaviors to adapt to changing contexts at runtime according to user-defined adaptation policies (rules).
- *Reactive actuators* are the actuators constructed at the system synchronization phase to process received data from peer programs. They reactively change their behaviors according to the active message header of the received data packet.

The rest of this paper is organized as follows. Section 2 covers the related work. Section 3 presents the MARCHES reflection model and system architecture. In Section 4, we theoretically analyze the reconfiguration time of MARCHES and compare it with peer research, followed by the system implementation and experiment validation in Section 5. The paper concludes with Section 6.

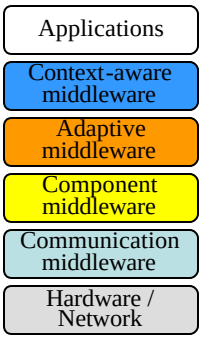


Fig. 2. Middleware layers.

2. Related Work

Middleware has been a critical technology for developing DRE systems because it can mask the heterogeneity of the underlying environment and simplify the task of programming and managing applications. It can be categorized into multiple layers (Fig. 2) based on the various functions provided for DRE systems.

Communication middleware focuses on integrating distributed computing systems to serve as a unified resource to reduce the application development cost. Early stage middleware, like CORBA, DCOM, and Java RMI, is built on Remote Procedure Call (RPC) to abstract the low-level TCP/IP communication details and replace the communication interface with a local procedure call or function invocation.

Component middleware, normally based on a component model (e.g. CORBA Component Model [17]), enables reusable service components to be organized, configured, and deployed for developing applications efficiently and robustly. Component middleware provides standards for object implementations and interactions so that it can support generic service components and then reduce the complexity of software upgrades and increase the reusability and flexibility of distributed applications. Existing component middleware contains both reusable common services, e.g. optimization of resource consumption (OSA+ [18], ACE [19]), configurability (TAO [3], Zen [20]), reusability (nORB [22]) etc., and domain-specific services, e.g. OSEK/VDX [23] for vehicle applications and ARINC 653 for avionics.

Adaptive and reflective middleware [24][25] has the ability to inspect its internal states by providing a representation of its internals through a process called reification. It also allows the internals to be dynamically manipulated and reconfigured through a process called absorption, which changes its non-functional and functional behaviors. The non-functional behavior reconfiguration is realized by dynamically replacing or changing the non-functional components of the *middleware*, like security check and concurrency control, etc. The functional behavior reconfiguration is realized by reconfiguring the functional components of the *application* at runtime. Open ORB [5] provides both structural reflection for functional component reconfiguration and behavioral reflection for nonfunctional component reconfiguration. Dynamic TAO [6] is a reflective ORB based on a set of component configurators. The TAOConfigurator can inspect and dynamically change its nonfunctional behaviors.

Context-aware reflective middleware can measure application's situational contexts and adapt application behaviors to them at runtime. It may be further divided into QoS-enabled middleware and user-defined context-aware middleware. QoS-enabled middleware can dynamically measure application-specific QoS and provide QoS reservation or adaptation to guarantee the required QoS, e.g. MUSIC [7], CIAO [12], Qoskets [15], and QuO [26]. User-defined context-aware middleware supports not only application QoS, but also any other user-defined contexts. MARCHES, MADAM [8], MobiPADS [9], and CARISMA [10] are some example systems that belong to this category.

QuO (Quality Objects) is a distributed object computing framework based on the CORBA model. It provides a QoS monitor and composes dynamic QoS provisioning capacity into DRE systems. QuO separates the QoS provisioning functionality from the application functionality; however, it still relies on ACE and TAO as it must use ORB based communication interfaces (e.g. TAO A/V streaming) and QoS tools (e.g GQoS and IntServ). MUSIC separates the self-adaptation concern from the business logic concern and delegates the complexity related to self-adaptation to generic middleware. It offers an adaptation-planning framework to evaluate the utility of alternative configurations in response to context changes, select a feasible one (e.g., the one with the highest utility) for the current context, and adapt the application accordingly. MADAM is a type of client/server based CARM for adaptive mobile applications. A master node (client) negotiates with slave nodes (servers) for an adaptation decision. It provides both reactive and proactive negotiation mechanisms for distributed adaptation decision. None of these frameworks provides any synchronization functionality; they assume that the adaptation has been constrained in safe conditions in advance. For example, the reconfiguration in QuO must be carefully studied so that the received data can still be understood by the receiver after reconfiguration.

MobiPADS is a policy- (or rule-) based CARM framework for mobile applications. It supports both middleware-layer and application-layer adaptations according to user-defined policies. A client middleware agent uses a communication channel to synchronize the application behaviors with a server middleware agent in a synchronous way whenever the architecture is reconfigured. The reconfiguration process includes operation suspension, buffer clearance, and chain-structure modifications. Because the initiator of the synchronization must be suspended until the system architecture of its own and other participants is reconfigured and the buffered data for previous architecture is processed, the reconfiguration time is in a range of seconds or even more according to the published experiment results. CARISMA employs a novel micro-economic approach that relies on a particular type of sealed-bid auction to handle the adaptation conflicts between distributed policies. The processing time of the conflict resolution algorithm includes communication time among peer agents for message exchanges and local computation time for context evaluation, bidding calculation, and solution set computation. This reconfiguration process is still synchronous and the conflict resolution algorithm must be invoked whenever a context is changed. Similar to these frameworks, MARCHES is also a policy-based CARM framework and focuses on the reconfiguration of stateless applications. MARCHES is different from existing work because it maintains multiple component chains and leverages the active messages to realize the synchronization in an asynchronous way. According to analysis and evaluations, MARCHES can significantly reduce the reconfiguration time and satisfy the responsiveness requirement of DRE systems. The preliminary results of MARCHES were published in [30]. In this research, we present a substantial extension of the system and thoroughly evaluate its performance based on a proposed analytical model and experiments.

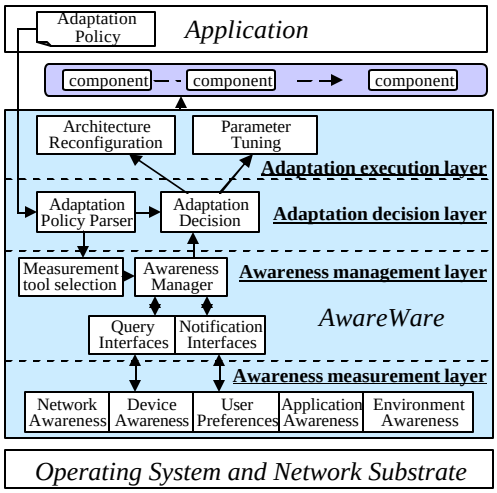


Fig. 3. System architecture of MARCHES.

```
<component cid="2002">
 <addr> D:\Masslets\JPEG.dll </addr>
 <name> Masslets.Compress.JPEG </name>
 <ctype> Masslet </ctype>
 <alias> COMPRESS </alias>
 <param pid="001">
  <name> SetCompressQuality </name>
  <vtype> Int32 </vtype>
  <value> 50 </value>
 </param>
 <interface iid="001">
  <name> PtrDataInput </name>
  <itype> Input </name>
  <Message> PDIBEventArgs </Message>
 </interface>
 <interface iid="002">
  <name> DataOutput </name>
  <itype> Output </name>
  <Message> JPEGEventArgs </Message>
 </interface>
</component>
```

Fig. 4. The component declaration in MARCHES.

3. System Architecture of MARCHES

MARCHES is located between the upper application layer and the lower operating system and network layer to monitor contexts and support application adaptations. It is peer-to-peer middleware with one middleware agent per application in each host. MARCHES consists of four major function layers as depicted in Fig. 3:

- **The awareness measurement layer** consists of individual measurement tools, which may measure context-awareness information about networks, devices, end-user preferences, application internal states, and physical environments.
- **The awareness management layer** hosts an awareness manager that communicates with the measurement layer through notification and query interfaces. It organizes and evaluates measured contexts based on event trees (called sensors) built on a hierarchical event notification model.
- **The adaptation decision layer** has a script parser and a decision engine. The script parser parses the adaptation policy script file defined by application developers based on a declarative language in an XML format. The decision engine takes the adaptation policy file as input, creates the awareness manager and a reconfigurator in the adaptation execution layer, and subscribes the actuators in the reconfigurator to the sensors in the awareness manager according to the adaptation policies. This allows the actuator to be triggered by context changes for reconfiguration according to the policies.
- **The adaptation execution layer** contains a reconfigurator to execute the behavior changes of functional and nonfunctional components. In this paper, we focus on the functional reconfiguration for improving the performance of DRE systems, which includes the component chain reconfiguration and component parameter tuning.

Between the middleware and application, there is another layer called the operation layer, in which various services are offered by software components. MARCHES supports application-specific components and standard third-party components based on its reflection model.

In this research, because we aim at improving reconfiguration efficiency of DRE systems, we focus on stateless applications and the reconfiguration of application-layer functional components. The proposed synchronization protocol can be combined with state-machine and model-based reconfiguration techniques to support the reconfiguration of stateful applications [16]. We also leave the reconfiguration of middleware-layer nonfunctional components, e.g. the concurrency, security, etc. as our future work, which can potentially be supported by MARCHES.

In summary, MARCHES is responsible for monitoring situational contexts that trigger adaptations, deciding when, where, and how to adapt application behaviors, and executing the adaptation policies specified by application developers at runtime.

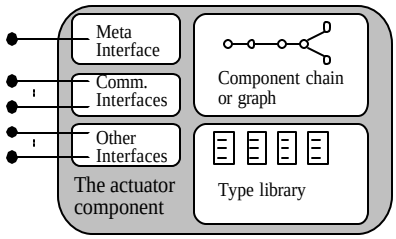
3.1 MARCHES Reflective Model

MARCHES supports both component-level and system-level reflection. The component-level reflection deals with the content and behavior of a given component via an interface metamodel, which provides a way to discover and access the interfaces of a software component. Thus, reflective components can be supported by MARCHES to incorporate new techniques and services and meet the upgrade and extension of DRE systems. The system-level reflection deals with the structure of the component connections via an architecture metamodel, which enables the discovery and operation of the current active component chain. The system-level reflection allows MARCHES to examine its internal states at runtime and dynamically reconfigure the application architecture to enhance its adaptability.

3.1.1 Components and component-level reflection

A MARCHES component is a function-independent reflective element that provides an interface metaobject. This interface metaobject allows a component to read its own metadata, extract the metadata from the component (called reification), and use that metadata to either inform the component user or modify the component's behavior (called absorption). By using the interface metamodel and component-level reflection, MARCHES can examine the types in a standard component, create new types at runtime, instantiate the types, and dynamically invoke properties and methods on the instantiated objects (called the late binding).

To incorporate a new software component in MARCHES, users need to describe the types, interfaces, and other attributes of the component in a system script file using the defined IDL (Interface Description Language), as shown in Fig. 4. There are three



(a)

```
interface IMetaActuator
{
   componentList get_components();
   connectionList get_connections();
   bool set_components(componentList compList);
   bool set_connections(connectionList connList);

   bool add_component(CMarchletObj marchlet);
   bool remove_component(string marchlet);

   bool connect_all_components();
   bool disconnect_all_components();

   bool connect_components(string senderObj, string senderInterf,
      string receiverObj, string receiverInterf);

   bool disconnect_components(string senderObj, string senderInterf,
      string receiverObj, string receiverInterf);

   Object get_component_parameter(string comp, string param);
   bool set_component_parameter(string comp, string param, Object value);

   EnumActuatorStatus get_active_status();
   bool activate();
   bool deactivate();
   ……
}
```

(b)

Fig. 5. (a) The MARCHES actuator architecture and (b) its meta-interface.

methods to identify a MARCHES component: 1) the exclusive component *name* for a registered system component, 2) the complete *address* for a local component, or 3) the desired *attributes* for a registered component in the component manager. The component type is declared in the *ctype* part and the *alias* is the name of the component used in the adaptation policy part of the script. The component can be specified by setting its parameters, which can also be reconfigured at runtime according to adaptation rules. It also provides some *interfaces*. The input and output interfaces can be bound together through connectors if they support compatible event messages and their connections can also be reconfigured at runtime.

There are two types of MARCHES components: reconfigurable functional components (namely *marchlets*) and extensible context-awareness components (namely *marchtools*).

Marchlets are the basic functional units to construct DRE systems. Each marchlet provides output and input interfaces for component assembly and communication based on the publish/subscribe model [29]. An output interface of a marchlet can be subscribed by message-compatible input interfaces of other marchlets and publish messages to them through connectors.

Marchtools, which measure and predict real-time context changes in MARCHES, are realized as reflective components to facilitate the reuse and extension of existing measurement tools. Marchtools act as the lowest event sources that can be subscribed by higher level event nodes and organized in a hierarchical way to build sensors. There is a special type of *marchtool*, called the function component, which supports user-defined functions to pre-process the results of measurement tools, e.g. getting the average value of the bandwidth in the last 5 minutes. A function component can subscribe to marchtools and process their raw data as input parameters through interfaces.

To better maintain and update MARCHES components, we have proposed a distributed service module, called the component manager, which accepts component registration and provides them runtime environments. A registered component can be identified by MARCHES through its attribute name and value pairs. The major functions of the component manager include component evaluation that is used to discover and utilize components not considered when the systems were designed. [26], component migration that is used to migrate required components from peer agents when the components are not available locally [27], and virtual connection that is used to process high workload tasks in a resource-limited device by connecting to a physical component hosted in a powerful server[28]. Due to page limitation, these functions will not be discussed in this paper.

3.1.2 Reconfigurator and system-level reflection

The MARHES reconfigurator contains multiple actuators and provides interfaces to manipulate the actuators so that the application behaviors can be reconfigured. The actuators are designed as reflective components to support MARCHES system-level reflection. Each actuator (see Fig. 5a) contains a component chain for processing application data, a type library for browsing the component types, and a meta-interface exemplified by Fig. 5b in C#. The meta-interface provides the access to its underlying meta-information and internal states (reification), such as the structure of component connections, the actuator status (active/inactive), etc. By accessing the meta-interface, the reconfigurator can change the actuator's meta-information that leads to a change of the actuator implementation (absorption), including the structure modification of component connections and component parameter modifications.

3.2 Awareness Measurement Layer

To support adaptation, DRE systems need to be aware of their running contexts. In this paper, awareness is defined as the contextual information of DRE systems. Most existing context-aware middleware frameworks have the functionality to detect a certain context. Our efforts in awareness measurement focus on integrating existing tools that are publicly available and providing mechanisms for application developers to specify and customize these tools in the XML format.

3.2.1 Measurement tools

Measurement tools in MARCHES are implemented as reflective components, which can be declared in the script file, and then loaded and instantiated by MARCHES to measure interested contexts. For example, the real-time QoS monitoring tool and the Mobile Service Testing and Measurement Tool (MOSET) [13] can be declared in the *Marchtools* section of the script file, see Fig. 4, to measure application-related QoS. Marchtools can also be reconfigured to realize feedback control.

For awareness data that is unavailable from local measurement tools or beyond the middleware knowledge, like the remote

information, the measurement is separated into two steps based on an information manager (IM). Awareness providers, like remote measurement tools and applications, send the awareness results to the IM. Marchtools then retrieve the data from the IM through pull or push methods. By pulling, marchtools explicitly query awareness data. By pushing, the IM pushes data to subscribed marchtools when pre-defined conditions are satisfied.

3.2.2 Context-awareness categorization

MARCHES categorizes the context awareness data in five categories listed in Table 1. Among these five categories, network awareness has continuously stimulated the interest in research and industry communities to provide reliable network-awareness measurement tools. Device awareness data, such as the CPU power, display size, memory capacity, display refresh rate, and battery consumption, may be measured through system APIs. User awareness can be collected in an explicit or implicit technique. In an explicit approach, users can specify their preferences through graphical user interfaces. In an implicit approach, measurement tools identify users' preferences by using machine learning agents. The environment awareness is measured by physical sensors.

Table 1. Categories of MARCHES context-awareness

Network-Awareness	Network characteristics and its measurements
Device-Awareness	Capacity measurements of a particular device
Application-Awareness	Internal states of an application or application required QoS
User-Awareness	User specified preferences for the quality of the service
Environment-Awareness	Environmental measurements by wireless sensor networks

3.3 Awareness Management Layer

The awareness manager in the management layer aims to organize and evaluate the contexts measured from the awareness measurement layer. In DRE systems, data from multiple awareness categories may be needed for context evaluations. For example, a DRE system involving video transmissions may rely on the information of both local hardware resource and network bandwidth to select a proper compression strategy. The first difficulty of the awareness management is that the communication network among marchtools cannot be fixed in advance since it is impossible to specify the marchtools that are used by applications at middleware design-time. Fortunately, this difficulty can be solved by the component-level reflection introduced in section 3.1. With the reflection model, users only need to specify the interfaces and parameters of marchtools in a script file and the awareness manager will set up the communication network at run-time based on the subscribe/notification model.

The second difficulty is that the awareness manager should get sufficient information for accurate adaptation with as few messages as possible to fit the limited resource of DRE systems. To solve this difficulty, a binary tree based hierarchical event notification model (see Fig. 6) is proposed for conditional subscriptions. This allows context events to be organized and integrated in a tree structure to construct a sensor that only

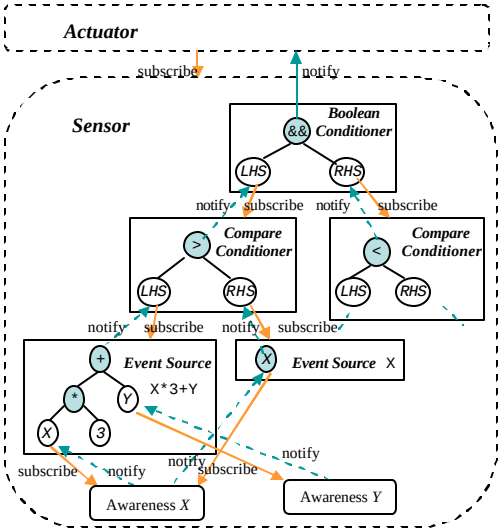


Fig. 6. The event notification model.

monitors and evaluates required contexts and triggers reconfigurations at runtime when its conditions are satisfied.

Each node in the event tree contains a *conditioner*, a left hand side (*LHS*), and a right hand side (*RHS*). There are two types of conditioners: the *compare conditioner* and the *Boolean conditioner* that perform comparison and Boolean operations on the LHS and the RHS. The LHS and the RHS can subscribe to the conditioner of a lower-layer event node or an event source. The event source can be a constant value, single context awareness, or an awareness expression. The expression is also built on a binary tree structure, in which each node has an operator, a LHS, and a RHS. Therefore, all the contexts are organized in a hierarchical way to form a sensor. An upper-layer event node or an actuator can subscribe to a lower-layer node as a listener, and only be notified when the conditions of the lower-layer node are satisfied. This structure minimizes the message exchanges in complex sensors.

To improve the efficiency of sensors, the hierarchical event tree is constructed based on the Modified Directed Acyclic Graph (MDAG). That is, before creating a new event node, it checks whether an identical node or an inverse node already exists. Event node *a* is defined as the inverse node of *b* if *a* and *b* have the same event source and comparison value, but inverse comparison operators. For example, the inverse event of " Min(AVI_CPU, 10) < 1.0" is "Min(AVI_CPU, 10) =1.0".

To use the event model to identify interested contexts, DRE system developers or end users declare corresponding sensors in a script file. The example shown in Fig. 7a means when the average bandwidth during the last 5 seconds is greater than 10*Mbps* and less than 20*Mbps*, the sensor notifies its subscribed actuators. To facilitate the configuration of the sensor script, the MARCHES Generator, a tool with Graphic User Interface (GUI), has been developed to transfer a sensor defined in the advanced language (Fig. 7c) to a XML script (Fig. 7a) according to the operator mapping (Fig. 7b). More details about the MARCHES Generator will be discussed in the section 3.4.

3.4 Adaptation Decision Layer

The adaptation decision layer contains a decision engine and a script parser. The decision engine takes the script file as input,

```
<sensor>
 <event>
  <otype> And </otype>
  <lhs>
   <event>
    <otype> GT </otype>
    <lhs>
     <expr> Ave(AVI_BW, 5) </expr>
    </lhs>
    <rhs>
     <expr> 10 </expr>
    </rhs>
   </event>
  </lhs>
  <rhs>
   <event>
    <otype> LT </otype>
    <lhs>
     <expr> Ave(AVI_BW, 5) </expr>
    </lhs>
    <rhs>
     <expr> 20 </expr>
    </rhs>
   </event>
  </rhs>
 </event>
<sensor>
```

(a)

Script Operator	Development Operator
GT	>
GE	>=
LT	<
LE	<=
NE	<>
EQ	==
And	&&
Or	||

(b)

```
Ave(AVI_BW, 5)>10 &&
Ave(AVI_BW, 5)<20
```

(c)

Fig. 8. A sensor example (a) the sensor declaration in an XML script file, (b) the mapping table of script operator to development operator, and (c) the sensor declaration in user development tool.

creates the awareness manager in the awareness management layer and the reconfigurator in the adaptation execution layer, and subscribes the actuators in the reconfigurator to the sensors in the awareness manager according to the adaptation policies. This allows the actuator to be triggered by changing contexts for reconfiguration according to user-defined policies.

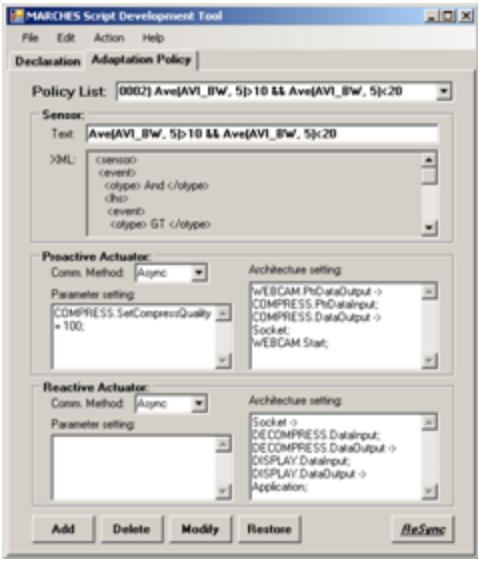
The script parser parses the application script file, which customizes the application configuration and adaptation policies based on a declarative language in the XML format. In particular, the script file can be divided into a declaration part and an adaptation-rule part (as shown in Fig. 8). The declaration part declares all components (marchlets and marchtools as shown in Fig. 4) used in a local program and the middleware agent. According to the declaration, MARCHES loads and instantiates

```
<Marchlets>  ...  </Marchlets>
<MarchTools>  ... </MarchTools>

<Rules>
 <rule>
  <sensor>   ...  <sensor>

  <Actuator type="proactive" sync="Async">
   <SetParam>
    COMPRESS.CompressQuality = 70;
   </SetParam>
   <SetArch>
    GRAB.PtrOutput -> COMPRESS.PtrInput;
    COMPRESS.StreamOutput -> SEND;
    Grab.Start;
   </SetArch>
  </Actuator>

  <Actuator type="reactive" sync="Async">
   <SetArch>
    RECEIVE -> DECOMPRESS.StreamInput;
    DECOMPRESS.StreamOutput -> DISPLAY.Input;
   </SetArch>
  </Actuator>
 </rule>
 ...
</Rules>
```

Fig. 7. An XML script file example.

the components through the reflection model. The adaptation-rule part contains adaptation policies and each policy can be further separated into a sensor, a proactive actuator, and an optional reactive actuator. A sensor section can be parsed by the event interpreter to build a sensor (as shown in Fig. 7) that monitors contexts and accepts the subscription of the proactive actuator declared in the proactive actuator section. The proactive actuator contains the system architecture information that is used to update the actuator internal states by the reconfigurator when it performs reconfiguration actions. Therefore, the system behaviors can be dynamically adapted to context changes through the system-level and component-level reflection (respectively, architecture reconfiguration and parameter tuning). The reactive actuator section describes the meta-information of an actuator in peer agents that processes the received data from the proactive actuator, so that the behaviors of the proactive and reactive actuators can be synchronized in distributed systems. The script

(a) Component declaration



(b) Adaptation policies

Fig. 9. The MARCHES script file development tool.

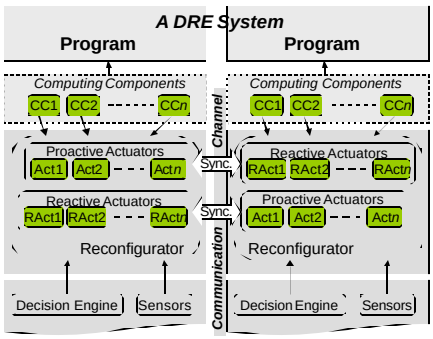


Fig. 10. The MARCHES reconfigurator.

example in Fig. 8 shows that the proactive actuator in the sender agent of a video transmission application contains three components: GRAB, COMPRESS, and SEND that are connected in a sequence. The reactive actuator described in the same policy contains the meta-information of three components as well: RECEIVE, DECOMPRESS, and DISPLAY. The receiver agent will construct the reactive actuator based on the meta-information received through the synchronization process.

The MARCHES Generator can facilitate users in generating script files. As shown in Fig. 9, the GUI tool allows users to manipulate both the component and policy configuration and runtime reconfiguration interactively. Furthermore, the tool supports the advanced language for describing event sensors and the *re-sync* function that can re-synchronize the local agent with peer agents when adaptation policies are modified at run-time.

3.5 Adaptation Execution Layer

The reconfiguration process of DRE systems consists of two steps: local behavior changes triggered by context changes and distributed behavior synchronization to synchronize local behaviors with the changed behaviors of other programs.

3.5.1 Local behavior reconfiguration

For traditional reflective middleware, there is only one component chain (or functional path) in each program. The reconfiguration process is to modify the chain structure. For the video transmission example, the original chain of the sender agent contains two components: GRAB and SEND, as shown in Fig. 1a. If the chain is reconfigured to contain three components: GRAB, COMPRESS, and SEND for adaptation, the reconfiguration process of the sender agent has the following steps:

- The sender agent stops its application workflow and stores component states into parameter lists;
- The sender agent clears buffered data that are not processed or transmitted to distributed peer agents;
- The sender agent disconnects the GRAB and SEND components and reconnects the GRAB, COMPRESS, and SEND components in sequence;
- The sender agent communicates with peer collaborative agents to synchronize the modified structure with them;
- Peer agents take the same steps: stop current workflow, clear buffered data received from the sender agent for old structure, and adjust component chains by disconnecting RECEIVE and DISPLAY components and reconnecting RECEIVE, DECOMPRESS, and DISPLAY components;
- The sender agent restores the states of the new component chain and restarts the application workflow.

The above reconfiguration process is synchronous and repetitive for each reconfiguration process. It is inefficient because the sender agent has to be suspended until all peer agents finish their corresponding reconfiguration, and all buffered data for previous structure are cleared.

By contrast, MARCHES supports multiple component chains as shown in Fig. 1b. Each component chain is located in an actuator that is subscribed to an event sensor (see Fig. 10). When contexts change and trigger a new sensor, the sensor will notify the decision engine for the reconfiguration by switching active and inactive actuators. There is one and only one active actuator that processes application data. For the above example, there are two chains in the sender agent: the active chain *i* contains two components: GRAB and SEND and the inactive chain *j* contains three components: GRAB, COMPRESS, and SEND. The reconfiguration of the sender agent has the following steps:

- The sender agent deactivates the current active actuator that contains chain *i* by suspending its workflow, storing run-time states, and disconnecting its components;
- It activates the target actuator containing chain *j* by connecting its components, restoring states, and resuming its workflow.

To reduce resource consumption, an actuator only maintains a chain of references, which point to marchlet instances, and a customized parameter list for each reference to store component runtime states. The proposed reconfiguration process is asynchronous and efficient because it does not require peer agents to synchronously reconfigure their structure and no buffered data needs to be cleared. The peer agents only synchronize their architecture on demand when their received data cannot be processed by existing reactive actuators.

3.5.2 Distributed behavior synchronization

Based on the multiple-chain based architecture, an active-message based synchronization protocol is designed to coordinate reconfigured behaviors in an asynchronous way. The idea of the proposed asynchronous protocol is that each middleware agent constructs the reactive actuators for all peer agents when the middleware starts up, and activates one of them to process received application layer packets according to the active message header attached in the packets. This initialization has the following steps, as shown in Fig. 11.

- When the middleware starts up, proactive actuators of each agent are built based on the user-defined script file. Each proactive actuator is also associated with a middleware-assigned unique index and the meta-information of an optional reactive actuator.
- The middleware agent sends a *synchronization request* packet to peer agents, which contains the indices of proactive actuators and the meta-information of reactive actuators.
- After receiving the *synchronization request* packet, the peer agent constructs the reactive actuators according to the meta-information, each of which is associated with a unique index. (The agent will notify the component manager for component migration or virtual connection if the required components can not be identified locally.)

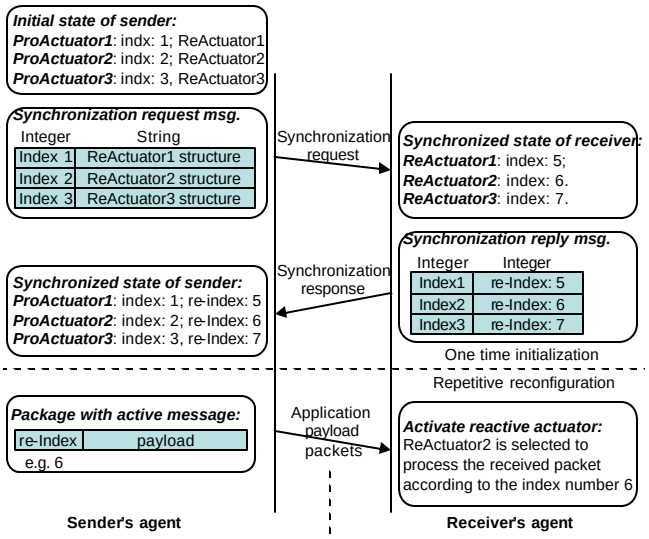


Fig. 11. The synchronization process.

- The receiver or the peer agent replies the sender a *synchronization response* packet that contains a set of index pairs, each of which contains an index of the proactive actuator and the index of the reactive actuator.
- The sender agent replaces the meta-information of each reactive actuator with its corresponding index received from the synchronization response packet.

The above-mentioned initialization is a one-time process. The middleware agent will then append the index of the reactive actuator, corresponding to the current active actuator, to the payload of each data packet as active message header. The peer agent receiving the data packet activates the reactive actuator indexed by the received index to process the data packet correctly.

The active message based asynchronous synchronization protocol has four advantages: low overhead, short delay, high efficiency, and better robustness. In general, only the index of the reactive actuator needs to be stored in the active message header for each data packet. By using the asynchronous method, the system does not need to be paused in the synchronization process, which dramatically reduces the reconfiguration time. Furthermore, based on the information in the active message header, a peer agent can always process the received packets by choosing the correct reactive actuator and then no suspension for buffered data is needed, which makes the reconfiguration by our middleware efficient. Moreover, once the reactive actuators are constructed in the system initialization phase, the local agent reconfiguration does not require the availability of other agents and thus it is not affected by the network condition or the capacity of other agents. Therefore the robustness of the application is improved and the communication overhead is reduced.

3.5.3 Correctness of MARCHES synchronization

In MARCHES, every received packet needs to be processed correctly by the agent to choose the indexed reactive actuator and the application workflow should not be affected or interrupted by middleware errors. To prove the correctness of the proposed synchronization protocol, we assume that 1) all errors are detected as errors; 2) a synchronization process may fail due to network errors, but it succeeds with at least some probability $p>0$; and 3) both sender agents and peer receiver agents may fail during the data transmission. We split the proof into two parts: safety and liveness. Safety is defined as the fact that a protocol never produces an incorrect result, which in this case means a received packet can always be properly processed. Liveness is defined as the fact that an algorithm can continue forever to produce results, which in this case means the capability to continue forever to send new packets at sender agents and accept them at receiver agents.

MARCHES uses re-transmission(s) for the synchronization process to handle network errors. If the number of synchronization failures from a sender agent to a peer agent is larger than a threshold, the peer agent is removed from the sender's receiver list. A new agent can request to join the sender's receiver list by sending a *node join* message to initialize the synchronization or leave the sender's receiver list by sending a *node leave* message to remove corresponding indices from the sender. After synchronization, every proactive actuator of the sender agent has an index (active message header) for each reactive actuator of every peer agent. To prove that a receiver agent can always process a received packet correctly, we consider the following three cases.

1) The receiver agent can recognize the index in the received active message header of the packet and the index is correct. This case represents the normal situation. The peer agents can invoke the reactive actuator pointed by the index to process the packet payload.

2) The receiver agent cannot recognize the index in the received active message header due to errors. This case happens when the receiver agent stops unexpectedly and does not notify the sender to remove it from its receiver list. The receiver can continue to receive packets from the sender after it is reloaded, but all the indices in the packets cannot be recognized as its local reactive actuators have been cleared. In this case, the receiver agent sends a *re-synchronization* message to the packet sender to initialize a synchronization process. Every received data packet can then be processed correctly without deletion.

3) For the completeness of discussion we consider the third case that will not happen: the receiver agent may recognize the index, but the index is incorrect and points to a wrong reactive actuator that cannot process the packet payload correctly. This case could be a concern if one would assume the following situation: both sender agent *a* and sender agent *b* communicated with the same receiver agent *c* and later *c* stopped unexpectedly while it might still be in the receiver lists of *a* and *b*; when *c* restarted and only synchronized with *a*, it constructed a reactive actuator for *a*, which might be used to process packets received from *b* if the reactive actuator for *a* shared the same index with the reactive actuator previously constructed for *b*. However, such assumed situation will not happen due to the way the index is created. In fact, the index of a reactive actuator is generated by the receiver agent based on the IP address of the synchronization requester and the hash value of the actuator meta-information. When both sender agents *a* and *b* are located at the same host and two different actuators from *a* and *b* have the same hash value, we will add an UID, which is unique for every application with respect to the host, in the active message header to ensure that each reactive actuator

has a unique index. Therefore, a recognized index must point to a correct reactive actuator.

Based on the above analysis, we can conclude that a MARCHES agent can always process received packets correctly, which proves the safety of MARCHES.

Once the synchronization process is completed, a sender agent only needs local information for reconfigurations. It can then continue to process application data forever during its life time. Thus the liveness of MARCHES is proved as we have concluded that a peer agent can always process received packets correctly.

3.5.4 Policy modification at runtime

The target users of MARCHES are DRE system developers. However, end-user requirements must also be considered in customizing the middleware behavior as it is difficult to predict all desired adaptation policies in advance. The MARCHES Generator provides end-users with the GUI (shown in Fig. 9), through which their preferences can be captured by MARCHES to modify the policies at runtime. MARCHES supports a re-synchronization method for the runtime policy modification, in which the agent suspends its operations, clears data buffer, re-synchronizes the modified policies with peer agents, and resumes its operations.

3.6 MARCHES Application Development

MARCHES offers an effective approach to build adaptive DRE systems. To develop a new system or migrate an existing system from another middleware framework to MARCHES, developers need to provide required components, an XML-based script file, and an optional GUI program for UI systems. The first step is to create MARCHES components or migrate existing components to the MARCHES platform. MARCHES component model currently supports COM components and .NET assemblies. To support other types of components, special component wrappers need to be developed, which will be part of the component manager functionalities in future work.

The second step is to develop a script file that declares all required components, including functional components (marchlets) and measurement tool components (marchtools), and adaptation rules using the XML language. The details of the script file structure are presented in section 3.4 and a full example is decomposed into Fig. 4, Fig. 7, and Fig. 8 for component declaration, sensor declaration, and policy declaration separately.

There are two different methods to instantiate and invoke MARCHES agents for constructing UI-based and service-based DRE applications. An UI application can create and invoke a MARCHES instance in their UI program through MARCHES interfaces. Some example UI applications using MARCHES have been published on our website [11]. A service application without a UI program can be constructed implicitly through the script development tool. After a script file is composed, the application can then be started, paused or stopped through the "Action" menu of the script development tool.

MARCHES is reflective middleware and provides a set of reification interfaces to present its internals to applications. The internals include component-chain structure, component states, reconfiguration events, and context information etc., which can be used for application debugging and validation. The applications also have the ability to reconfigure the internals through the absorption interfaces provided by MARCHES so that application users can manually manipulate the application behaviors.

4. Evaluation by Analytical Models

In this section, we theoretically analyze the performance of MARCHES in terms of the one-time initialization time in the middleware startup phase and the repetitive reconfiguration time whenever the reconfiguration process is triggered by context changes. To verify the time efficiency of the proposed multi-chain structure and active message oriented synchronization protocol, we compare the reconfiguration time of MARCHES with that of MobiPADS and CARISMA. These context-aware reflective middleware frameworks can be fairly compared because:

1. All these three frameworks are policy based with predefined adaptation policies specified in a script file.
2. They all target stateless applications, which do not require the guarantee of application states or packet delivery sequence in the reconfiguration process.
3. They all consider the behavior synchronization problem for distributed applications. However, the synchronization protocols they employ are different. MobiPADS uses a communication channel for synchronization and suspends application operations in the reconfiguration process. CARISMA uses a micro-economic approach to handle the adaptation conflicts between distributed policies. MARCHES uses an active-message-oriented asynchronous method for synchronization to solve the behavior inconsistency.

In the comparison, we ignore the component and code migration among different middleware agents, which will impose the same overhead for each framework. Since the theoretical analysis is system and implementation independent, we can then fairly compare their performance.

4.1 Analytical Model

To compare the reconfiguration efficiency of MARCHES with that of MobiPADS and CRISMA, we use a unified model to formulate the reconfiguration time as the sum of the communication time among distributed middleware agents and the local computation time.

$$T = T_{communication} + T_{computation} \quad (1)$$

To simplify of the model and for fair comparison, we ignore the component migration time required by all systems, the transmission delay of control messages, which is much smaller than their propagation delay as the control message size is negligible, and other overhead, like socket buffering time, thread switch time, and internal message exchange time, which may be affected by different operating systems and programming languages. All the middleware agents use TCP three-way handshakes for each reconfiguration message exchange, which takes $1.5RTT$[1] for the connection establishment.

4.2 Reconfiguration Time

In MobiPADS, there is only one component chain, and the reconfiguration process involves three steps: (*i*) initializing reconfiguration, (*ii*) deleting components, and (*iii*) adding components. The reconfiguration time expressed by Eq. (4) in [9] is shown as:

[1] In this paper RTT is defined as the minimum round trip time that only contains the propagation delay and the processing delay while excluding the transmission delay.

$$T_{MobiPADS} = (b + g + d)/B + 2kn + 2m + 5.5RTT + C \quad (2)$$

where ß is the meta-chain size; *?* and *d* are the component request message size and component size for component migration; 2*kn* is the component initialization time; *m* is the deletion time; and *C* is other overhead. We further separate *m* into 1.5 *RTT* for message exchange and an operation suspension time [9], and ignore the component migration time, the component initialization time, which is very small (i.e. few milliseconds as shown in Fig. 14), and other overhead to follow our model. We then rewrite Eq. (2) using the notations shown in Table 2 as Eq. (3) to express the reconfiguration time of MobiPADS for comparisons with those of MARCHES and CARISMA in a unified model.

$$\begin{aligned} T_{MobiPADS} &= T_{communication} + T_{computation} \\ &= (7RTT + s\_chain/B) + t\_pend \end{aligned} \quad (3)$$

t_pend is affected by the number of buffered data and their processing time, and its value may vary for different applications. We set its default value as 300*ms* to match the experiment results shown in Fig. 13 of [9] for numerical evaluations. The meta-chain size *s_chain* is analyzed in Section 5.2 and we set its default value as 10*Kbits* for a chain with 10 components. The reconfiguration time of MobiPADS is depicted in **Error! Reference source not found.**.

Table 2. Parameter notation of reconfiguration time

Notation	Parameter
RTT	The minimum round trip time excluding the transmission delay
t_tcp	TCP socket establishing time
t_pend	Operation suspension time for component deletion
t_init	Initialization time for a component addition
n_add	The number of components to be added in a reconfiguration process
s_chain	The average size of a meta-chain
t_reso	The total local computation time of the conflict resolution algorithm in CARISMA
n_policy	The number of policies in an application
t_react	Reactive actuator construction time in MARCHES
t_conn	Connection time of two MARCHES components
t_rest	Restoration time of a MARCHES component
n	The number of components in a component chain
B	The average available bandwidth

In CARISMA, the reconfiguration conflict resolution process consists of the following steps: 1) service request, 2) local context evaluation and enabled policy selection, 3) the enabled policy exchange, 4) solution set computation and conflict detection, 5) bidding request and reply, 6) winning policy calculation, and 7) the winning policy broadcast. Steps 1, 3, 5, and 7 involve communications for message exchanges, and steps 2, 4, and 6 involve local computations for conflict resolution, which is related to the number of policies, contexts, resources, and conflicts. To compare CARISMA with MARCHES in terms of reconfiguration time, we use the simplest case of CARISMA with minimum overhead, which is that each policy contains only one context and one resource and there is no conflict. The total reconfiguration time of CARISMA can then be expressed as:

$$\begin{aligned} T_{CARISMA} &= T_{communication} + T_{computation} \\ &= 4RTT + t\_reso = 4RTT + f(n\_policy) \end{aligned} \quad (4)$$

We use the values of the conflict resolution time *t_reso* that are directly obtained from the Fig. 15 in [10]. The reconfiguration time of CARISMA is shown in Fig. 13.

In MARCHES, the one-time initialization time in the startup phase includes 2.5*RTT* communication time, where 1.5*RTT* is for the TCP connection and 1*RTT* is for synchronization message exchanges and the transmission delay of the meta-data for multiple component-chains stored in the synchronization request message. The initialization time is represented as:

$$\begin{aligned} INIT\_T_{MARCHES} &= T_{communication} + T_{computation} \\ &= 2.5RTT + n\_policy \times s\_chain/B \end{aligned} \quad (5)$$

The initialization time of MARCHES reconfiguration is related to the *RTT*, the number of policies, and bandwidth. We set the *RTT* as 100*ms* (referring [9]) to compute the initialization time on the number of policies and the bandwidth, as shown in Fig. 14.

In the repetitive reconfigurations after the one-time initialization, the reconfiguration time is the sum of the component assembly and restoration time that is related to the number of components in the component chain. The reconfiguration time is expressed as:

$$T_{MARCHES} = T_{computation} = 2n(t\_conn + t\_rest) \quad (6)$$

The coefficient of 2 is needed because the reconfiguration process is carried out at both the proactive actuator of the sender and the reactive actuator of the receiver. Fig. 15 shows the reconfiguration time obtained by the benchmark experiments, see Section 5.1.2.

In summary, the reconfiguration time of MobiPADS and CARISMA is typically in the range of seconds and related to the network condition and system complexity. For example, MobiPADS reconfiguration takes about 2*s* for 20*Kbps* bandwidth and 1.4*s* for 1*Mbps* bandwidth according to our theoretical analyses, which achieve the same results with the experiments in [9]. CARISMA conflict resolution time is about 1.2*s* for 10 policies and 1.7*s* for 20 policies, and the time grows exponentially with the number of contexts and conflicts, conforming to the empirical results in [10]. Furthermore, their robustness is affected by the reconfiguration because it requires the availability of all the related peer agents and its failure would cause the crash of succeeding data packets.

Although the initialization time of the proposed asynchronous method in MARCHES is similar to the reconfiguration time of the synchronous methods in MobiPADS and CARISMA, the MARCHES initialization is a one-time process while the reconfiguration of MobiPADS and CARISMA is a repetitive process. The repetitive reconfiguration time of MARCHES after initialization is significantly shorter than that of MobiPADS and CARISMA. Moreover, the local agent reconfiguration is not affected by the network condition or the capacity of other agents, and the received data packets can be processed asynchronously based on their active message headers. Thus the robustness of the system is improved.

5. Evaluation by Experimental Measurements

We have implemented MARCHES in C# for both Windows XP (WXP) and Windows Mobile 5 (WM5) systems using visual studio 2005 and encoded the script file using XML. The testbed consists of two PCs (Thinkpad-X60: Intel T2300 1.66GHz, 512MB, and WXP), two PDAs (Dell x51v: Intel XScale 624MHz, 64MB, and WM5), two Cisco routers (Cisco 3200), and two switches (Cisco 2900XL). The routers are connected back-to-back through serial ports so that the network bandwidth can be controlled through the HyperTerminal tool.

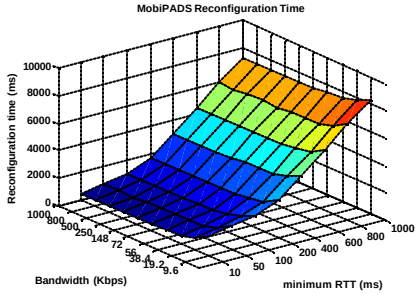


Fig. 12. MobiPADS reconfiguration time.

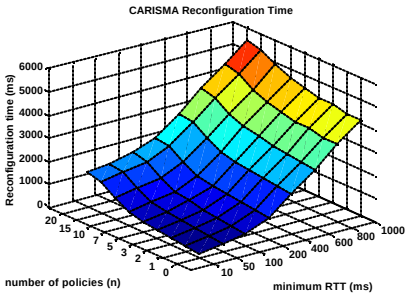


Fig. 13. CARISMA reconfiguration time.

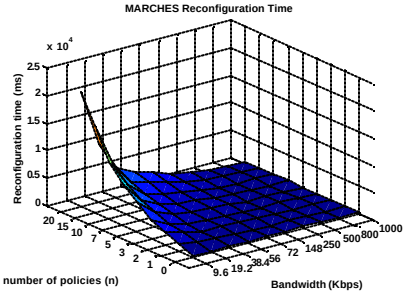


Fig. 14. MARCHES initialization time.

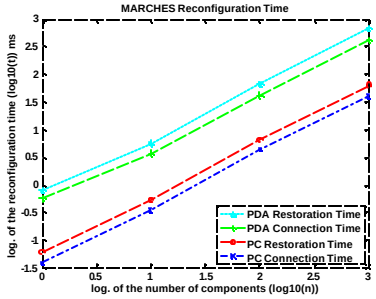


Fig. 15. MARCHES reconfiguration time.

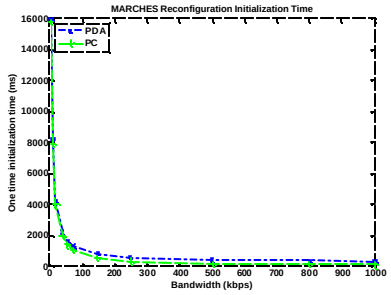


Fig. 16. Benchmark initialization time.

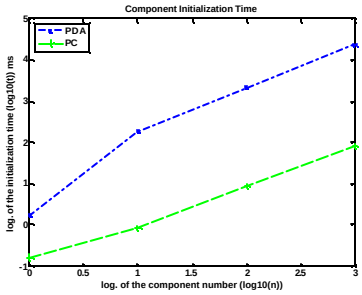


Fig. 17. Component initialization time.

The benchmark application consists of a set of simple components, each of which has a parameter interface, an input interface, and an output interface so that two components can be connected. Each adaptation policy has the same components in a single test and each reconfiguration process will disconnect and reconnect all the components and load one parameter for each component. The number of the policies and the number of components in each policy are varied in the experiments to simulate different applications. All the data are calculated based on the average of 10 test samples.

One of the goals of MARCHES is to reduce the reconfiguration time for DRE systems. However, the reconfiguration process introduces some performance cost, such as the active message header and extra resource consumptions for maintaining the multiple chains. Therefore it is important to check the feasibility and efficiency of using MARCHES. We evaluate its performance benefit and cost in this section in terms of the reconfiguration time, memory footprint, and scalability through benchmark applications on both PCs and PDAs.

5.1 Time Efficiency

5.1.1 One-time reconfiguration initialization time

The one-time initialization time of MARCHES is shown in Fig. 15 16. In the experiment, we set the RTT between the sender and receiver as 100*ms*. There are 10 policies in the benchmark application and each actuator meta-chain is 10*Kbit*. We then control the bandwidth by changing the clock rate of the router serial ports. The experiment results match well with the theoretical analysis shown in Fig. 14 with just slightly larger values because the unified model has ignored some processing overhead and control message transmission delay.

5.1.2 Repetitive reconfiguration time

The repetitive reconfiguration time is shown in Fig. 15. Because all the operations are executed in the same memory space and CPU process, it is in the range of several hundred microseconds to a few milliseconds. Moreover, the reconfiguration time is only determined by local hardware resources so that the time is very stable for every test.

5.1.3 Component initialization time

The component initialization time is defined as the time needed to load a component, check its types, and instantiate the component based on the encoded parameters. As shown in Fig. 17, the time is in the micro- to milli- second range.

We have also tested the event notification time of MARCHES sensors, which is another important metric to evaluate the responsiveness of MARCHES. Results show that it is in the microsecond level and much smaller than the repetitive reconfiguration time.

5.2 Memory Footprint and Scalability

In this experiment, we evaluate the local storage size and the run-time memory consumptions of MARCHES framework, components and actuators. We utilize the C# serialization function to serialize MARCHES and system objects and measure their run-time memory usage. Serialization means that objects are marshaled by value, that is, all their various member data are written out to the stream as a series of bytes. Therefore, we can use the length of the stream as the metric for memory consumption.

Table 3. Resource consumption by MARCHES.

Components	Windows XP system		Windows Mobile 5	
	Local file size	Run-time memory	Local file size	Run-time memory
Middleware	56KB	896KB	46KB	123KB
Empty marchlet	4KB	139Byte	4KB	74Byte
Simple marchlet	16KB	356Byte	5KB	147Byte
Simple marchtool	16KB	279Byte	4KB	94Byte

The local file size and run-time memory usage of MARCHES middleware and components are shown in Table 3 for both Windows XP (WXP) and Windows Mobile 5 (WM5) respectively. The run-time memory usage of the middleware is measured after initializing the system and before loading and instantiating any components. An empty marchlet is a reflective component containing no application-specific method or variable. A simple marchlet contains one input interface, one output interface, and 5 double-type parameters. Although we use very similar source code for both WXP and WM5, the run-time memory consumption is much different due to the code optimization in the mobile system.

Because MARCHES contains multiple actuators, it is important to analyze the overhead of the actuators. The memory consumption R is then expressed as:

$$R = \sum_i \left(\sum_j \left(\sum_k p_{ijk} + l_{ij} \right) + a_i \right) \quad (7)$$

where p_{ijk} (10Byptes) is the size of parameter k for marchlet j in actuator i; l_{ij} (12B) is the name and reference size of marchlet j in actuator i; and a_i (8B) is the index size of actuator i.

For a MARCHES agent that contains 5 actuators, 10 marchlets for each actuator, and 10 parameters for each marchlet, the resource consumption is 5640 *bytes* (≈ 5.5 *KB*).

In MARCHES, a middleware agent maintains not only local proactive actuators, but also reactive actuators built for remote peer agents through the synchronization. Thus, the memory consumption is closely related with the application scale. According to Eq. (7), the memory consumption R for a DRE system is then modified as:

$$R = \sum_t (R_t) \quad (8)$$

where t is the index of peer middleware agents.

For a DRE system that has 10 distributed programs and each program has a middleware agent described above, the memory consumption of the program is 5640 *bytes* × 10 (≈ 55 *KB*), which is still small compared to the capacity of most embedded devices.

5.3 Demo Applications and Releases

We have developed some adaptive DRE systems based on MARCHES, like a first responder system in PDA platforms and a distance education system in heterogeneous platforms (PCs, Laptops, and PDAs). The implementation of real applications demonstrates that MARCHES are easy to use, achieving fast responsiveness in reconfigurations, and supporting generic DRE systems. All the development and documents have been released on our website [11].

6. Conclusion

In this paper, we have described a context-aware reflective middleware framework called MARCHES to support adaptive DRE systems. MARCHES solves the critical issue of reconfiguration efficiency that has limited the adoption of traditional context-aware reflective middleware in DRE system development. Our solution includes a new adaptation structure of multiple component chains and a novel synchronization protocol using active messages to coordinate distributed reconfigurations asynchronously.

We have established a generic analytical model for fair comparisons of the reconfiguration efficiency of various context-aware reflective middleware. Besides theoretical analyses system performance of MARCHES has been evaluated using benchmark applications. The complete implementation of MARCHES and the benchmark applications allows us to test the feasibility and efficiency of MARCHES and gain insights into the DRE system design supported by it. The theoretical and experimental results demonstrate that (i) the reconfiguration time in traditional adaptive and reflective middleware is reduced by several magnitudes from seconds to hundreds of microseconds, (ii) the extra costs introduced by the multi-actuator architecture in MARCHES are extremely low, and (iii) the robustness and scalability are improved as well in MARCHES compared with traditional middleware.

The following future work of MARCHES will be explored.

- **The MARCHES extension for stateful applications**: The proposed synchronization protocol can be combined with state-machine and model based reconfiguration techniques to improve the reconfiguration efficiency of a state application, like the GSM-Oriented coding application [16].
- **MARCHES component model:** MARCHES supports COM components and .NET assemblies so far. We will extend the component manager to support more component models and components like CORBA components and JAVA Beans etc.

7. Acknowledgements

This material is based upon work partly supported by the National Science Foundation (NSF) under Grant No. 0438300 and the Pennsylvania Department of Community and Economic Development (PA-DCED) via the PITA (Pennsylvania Infrastructure Technology Alliance) program. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF and the PA-DCED/PITA. The authors would also like to thank Lisa Frye for proofreading this paper.

8. References

[1] Tarek, A., Christopher, D.G., Raj, R., John, A.S., 2006, Distributed Real-time and Embedded Systems Research in the Context of GENI. NSF Workshop on Distributed Real-time and Embedded Systems.

[2] Judith, A.C., Audrey, E.T., 1998. A Management Guide to Software Maintenance in COTS-Based Systems. MP 98B0000069, The MITRE Corporation, Bedford, MA.

[3] Douglas, C.S., David, L.L., and Sumedh, M., 1998. The Design of the TAO Real-time Object Request Broker. Computer Communications, 21(4): 294–324.

[4] Hu, J., Gorappa, S., et. al., 2007. A Lightweight Component Middleware Framework for Composing Distributed, Real-Time, Embedded Systems with Real-Time Java. In Proc. ACM/IFIP/ USENIX 8th Int'l Middleware Conference (Middleware'07). Lecture Notes in Computer Science, Vol. 4834, pp. 41–59.

[5] Gordon, S.B., Geoff, C., Anders, A., 2001. The Design and Implementation of Open ORB 2. IEEE Distributed Systems Online 2(6).

[6] Fabio, K., Manuel, R., Ping, L., et. al., 2000. Monitoring, Security, and Dynamic Configuration with the DynamicTAO Reflective ORB. In Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'00).

[7] Romain, R., Paolo, B., et. al., 2009. MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments. Self-Adaptive Systems, LNCS 5525, pp. 164–182.

[8] MADAM Consortium, 2006. Specification of the MADAM Core Architecture and Middleware Services. Final report within the 6th Framework Programme, Priority 2.3.2.3.

[9] Alvin, T.C., Siu-Nam, C., 2003. MobiPADS: A Reflective Middleware for Context-Aware Mobile Computing. In IEEE Transaction on Software Engineering, 29(12).

[10] Licia, C., Wolfgang, E., Cecilia, M., 2003. Carisma: Context-aware Reflective Middleware System for Mobile Applications. In IEEE Trans. on Software Engineering, 29(10): 929–945.

[11] MARCHES homepage, accessed on August 10, 2010. http://marches.cse.lehigh.edu/.

[12] Nanbor, W., Douglas, C.S., Michael, K., 2003. Towards an Adaptive and Reflective Middleware Framework for QoS-enabled CORBA Component Model Applications. IEEE Distributed System Online, Special Issue on Reflective Middleware.

[13] Palola, M., Jurvansuu, M., Korva, J., 2004. Breaking Down the Mobile Service Response Time. In Proceedings of IEEE International Conference on Networks (ICON '04), Singapore, pp. 16-19.

[14] Ron, B.N., 1995. CORBA: A Guide to Common Object Request Broker Architecture. McGraw-Hill, Inc.

[15] Praveen, K.S., Joseph, P.L., et. al., 2004. Component-based Dynamic QoS Adaptations in Distributed Real-time and Embedded Systems. In Proc. the International Symposium on Distributed Objects and Applications (DOA '04), vol. 3291, pp. 1208–1224.

[16] Ji, Z., Betty, H.C., 2006. Model-Based Development of Dynamically Adaptive Software. In International Conference on Software Engineering (ICSE 2006).

[17] Object Management Group, 1999. CORBA Component Model Joint Revised Submission. OMG Document orbos.

[18] Uwe, B., Aurelie, B., Florentin, P., Etienne, S., 2002. Distributed Real-Time Computing for Microcontrollers - The OSA+ Approach. In Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, page 169. IEEE Computer Society.

[19] Schmidt, D., Huston, S., 2001. C++ Network Programming: Resolving Complexity with ACE and Patterns. Addison-Wesley, Reading, MA.

[20] Raymond, K., Douglas, C.S., 2002. Carlos, O., Towards Highly Configurable Real-time Object Request Brokers. In Proceedings of ISORC'02, pp. 437–447.

[21] Zinky, J.A., Bakken, D.E., Schantz, R., 1997. Architectural Support for Quality of Service for CORBA Objects. Theory and Practice of Object Systems, 3(1), pp. 1-20.

[22] Venkita, S., Guoliang, X., Christopher D., and Cytron, R., 2003. The Design and Performance of Special Purpose Middleware: A Sensor Networks Case Study. Technical Report WUCSE-2003-6, Washington University.

[23] OSEK Comitee, OSEK/VDX homepage, accessed on August 10, 2010. http://www.osek-vdx.org/.

[24] Garbinato, B., Guerraoui, R., and Mazouni, K.R., 1995. Distributed Programming in GARF. In the Proceeding of the ECOOP Workshop on Object-Based Distributed Programming, Springer-Verlag, Kaiserslautern, Germany, pp. 225-239.

[25] McAffer, J., 1993. Meta-level Programming with CodA. In Proceedings of the European Conference on Object-Oriented Programming (ECOOP).

[26] Justin, M.P., Hubert, P., Umar, S., Grace, C., Chris, T., Steve, W., 2008. Structured Decomposition of Adaptive Applications. In Proceedings of the 6th IEEE International Conference on Pervasive Computing and Communication (PerCom).

[27] Stefanos, Z., Cecilia, M., 2006. The SATIN Component System—A Metamodel for Engineering Adaptable Mobile System. In IEEE Trans. on Software Engineering, 32 (11).

[28] Radu, L., Atul, P., 1999. DACIA: A Mobile Component Framework for Building Adaptive Distributed Applications. Technical Report CSE-TR-416-99, University of Michigan.

[29] Ranganathan, A., Campbell, R.H., 2003. An infrastructure for context-awareness based on first order logic. Personal Ubiquitous Computing, 7(6): 353-364.

[30] Liu, S., Cheng, L., 2008. Active Message Oriented Adaptation Middleware for Collaborative Applications in Heterogeneous Environments. In Proc. the 2008 IEEE International Conference on Communications (ICC 2008), pp. 1866-1870.

[31] Kasten. E.P., McKinley, P.K., 2001. Adaptive Java: Refractive and Transmutative Support for Adaptive Software. Technical Report MSU-CSE-01-30, Department of Computer Science and Engineering, Michigan State University.