# The PADRES Distributed Publish/Subscribe System

E. Fidler H.-A. Jacobsen G. Li S. Mankovski
Middleware Systems Research Group Cybermation, Inc.
University of Toronto Markham, Ontario, Canada

**Abstract.** Workflow management systems are traditionally centralized, creating a single point of failure and a scalability bottleneck. In collaboration with Cybermation, Inc., we have developed a content-based publish/subscribe platform, called PADRES, which is a distributed middleware platform with features inspired by the requirements of workflow management and business process execution. These features constitute original additions to publish/subscribe systems and include an expressive subscription language, composite subscription processing, a rule-based matching and routing mechanism, historc, query-based data access, and the support for the decentralized execution of business process specified in XML. PADRES constitutes the basis for the next generation of enterprise management systems developed by Cybermation, Inc., including business process automation, monitoring, and execution applications.

## 1 Introduction

This paper describes a new distributed content-based publish/subscribe system called PADRES and presents a powerful business-based approach for workflow management built on the PADRES messaging substrate. The research is part of the Toronto Publish/Subscribe System research efforts [19, 14, 2, 18, 22]. Requirements for PADRES were determined in collaboration with Cybermation, Inc., which has developed the ESP Workload Manager and ESP Espresso products. The PADRES system was engineered to support several novel concepts in p/s that are necessary in the workflow management context. We envision that the next generation of workflow management solutions will certainly be built on asynchronous messaging substrates like PADRES. Matching in PADRES is performed using a powerful rule engine to enable composite subscriptions. The matching engine is fully distributed across the federation. Access to previously-published (historic) data is another novel feature useful for monitoring and auditing. The business-based approach to workload management provides an efficient means of workflow scheduling and supports dynamic execution patterns.

A p/s system [6] is comprised of information producers who publish and information consumers who subscribe to information. It provides a simple and effective method for disseminating data while maintaining a clean decoupling of data sources and data sinks. Most existing p/s systems are limited by subscription languages that are too simple to describe complicated interests, and the lack of a high-level view of events in the system. Many existing p/s systems [3, 16, 20] restrict subscriptions to single events, and thus lack the ability to express interests in the occurrence of complicated global event patterns. Subscribers may be overwhelmed by large numbers of primitive events that are not valuable unless a particular event pattern is detected among the primitive events. An additional restriction on existing p/s systems is that subscribers can only

subscribe to future data; they can not receive data published before their subscriptions were issued. As far as we know, there has been no research on historic data access in p/s systems or on composite subscription matching in distributed broker networks.

Applications of p/s technology are numerous; selective information dissemination, location-based services, network management, and workflow management are a few examples. There has, however, been little attention given to workload management and business process execution in the p/s literature. Workflow management is a fundamental component of any data-centric operation, and any gains made in this area have far reaching impact and benefits for enterprises. Workflow management solutions have existed for a number of decades. The traditional approach to workflow management is based on using a central server to coordinate and schedule tasks. This architecture is inherently limited. The centralized monitoring and control point constitutes a single point of failure in the system. It forms a bottleneck for system monitoring and control, and can significantly affect overall application operation and scheduling performance. Moreover, a centralized approach is limited to workflow management in a single administrative domain. Resources distributed across administrative boundaries (i.e., across multiple data centres) can not be federated with ease, and distributed jobs are hard to handle. This is particularly an issue not only in the e-commerce sector, but also in the emerging Grid computing model, where peak load management and on-demand dynamic resource allocation will become a challenge for workflow management. The centralized solution may be feasible if the manager itself is executed on a mainframe computer, but most companies are instead relying on clusters of smaller machines. The geographical distribution of jobs calls for a distributed middleware architecture and p/s is an ideal candidate. A federated system allows the clustered workload management system to achieve the necessary high levels of performance and reliability. As a key aspect of a federated system, flexible network links and efficient, scalable routing algorithms are necessary for the federated system.

The rest of this paper is structured as follows. Section 2 contains background material and related work. Section 3 introduces PADRES, including its network architecture, internal broker, composite subscription and historic data access features. The content-based p/s system acts as a substrate for distributed workflow management systems, which are discussed in detail in Section 4. The deployment and execution of a workflow are presented. Section 5 shows the evaluation results of PADRES and the workflow management system. Conclusions are given in Section 6.

## 2 Background and Related Work

**Publish/Subscribe** In the p/s paradigm, information producers submit data as publications to the system and information consumers indicate their interests by submitting subscriptions. A subscription has a notification set, which is a set of potential publications that would match the subscription. On receiving a publication, the broker determines the subset of matching subscriptions and notifies the appropriate subscribers. This paradigm has found wide-spread use in applications ranging from selective information dissemination to network and distributed application management, such as workflow management.

While p/s was first implemented in centralized client-server systems, current research focuses mainly on distributed versions. The key benefit of distributed p/s is the natural decoupling of publishers and subscribers. Since the publishers are unconcerned with the potential consumers of their data, and the subscribers are unconcerned with

the locations of the potential producers of interesting data, the client interface of the p/s system is simple and intuitive.

**Content-based Routing** Content-based p/s systems typically utilize *content-based routing* in lieu of the standard address-based routing. Since publishers and subscribers are decoupled, a publication is routed towards the interested clients without knowing specifically where those clients are and how many such clients exist. Effectively, the content-based *address* of a subscriber is the set of subscriptions issued. There are other approaches dealing with content-based routing. For instance, Hermes [20] implements a content-based notification delivery system on top of a peer-to-peer overlay network. SIENA [3] is a notification service that uses covering-based routing, and REBECA [15] discusses merging-based routing. These two techniques can also be applied in PADRES. In addition to publications and subscriptions, content-based routing networks can use *advertisements* [15, 3], which are indications of the data that publishers would publish in the future. Advertisements are used to form routing trees along which subscriptions are propagated. Without advertisements, subscriptions must be flooded throughout the network.

**Composite Subscription** In traditional p/s systems, each publication match is an independent stateless operation. Composite subscriptions, which consist of primitive (*atomic*) subscriptions, are analogous to composite event detectors in event-processing systems. A composite subscription is matched only after all component *atomic* subscriptions are satisfied. That is, a composite event pattern in the p/s system occurs and is detected. Composite events are a key concept in the event-processing context and have not received much attention in the p/s literature. Some distributed p/s architectures such as Hermes [20] and Gryphon [16] provided only parameterized primitive events, and lacked the composite event detection module in the internal p/s system, which is responsible for disseminating events. SIENA [3] supports restricted event patterns, but it does not define a complete pattern language. There has been some work in building a distributed composite event detection framework on top of p/s systems [21], but the detection is not in the p/s layer.

**Workflow Management** A workflow management system performs coordinated execution of computer *jobs*. The job execution *agents* are clients in a p/s system, acting as both subscribers and publishers. The jobs, which are executed according to predefined logic, are composed into *virtual workflows*. A workflow is also called an *application* in the rest of this paper. The workflow management system performs two key tasks: job execution and job monitoring. Job execution is simply the invocation and control of jobs using remote methods such as RPC and Web Services. Job monitoring involves maintaining a set of status information for each executing or previously executed job. Workflow management is one of the earliest suggested applications of expressive triggers in the DBMS [5]. Some commercial products, such as Oracles's *Workflow Builder* [17] and Informix's *Media360* package [11] both provide tools that generate triggers from workflow specifications. Hagen *et al.* [9] describe an approach for event-based communication for workflows cooperating in a consumer-producer relationship. This is a centralized management by maintaining a *job dependency graph*. The Business Process Execution Language (BPEL) [10] is a standard for business process execution systems, and it provides a language for the formal specification of business processes and business interaction protocols. The JBPM [12] project, built for the JBoss middleware platform, is an example of a workflow management and business process execution system.

Cybermation Inc. has working on building workload and job scheduling solutions for twenty years. Their experience on ESP Workload Manager and ESP Espresso prod-

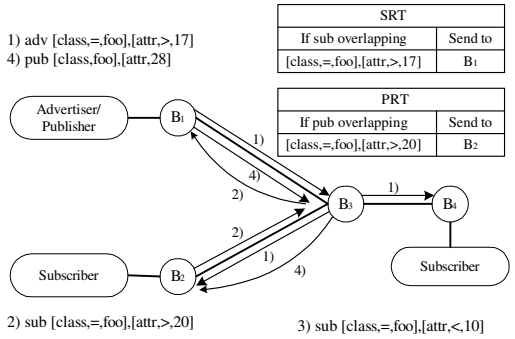


Figure 1: PADRES Network Architecture

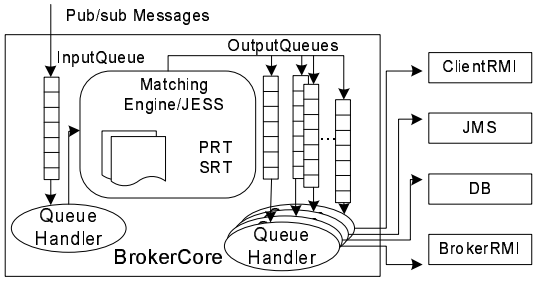


Figure 2: PADRES Broker Internals

ucts gives us the cherishable user feedback. Workflow management requires several features not normally found in a p/s system. The job monitoring function requires access to data regarding jobs which have previously executed. The ability to subscribe to publications sent before the subscription is not present in any existing p/s system. The subscription language is limited to primitive subscriptions, which cannot describe complicated job dependencies in a workflow. In fact, subscriptions that indicate the publication time(s) of interest are generally not supported. The scalability and wide-area execution requirements of workflow management indicate a distributed architecture. A distributed p/s system allows for better scalability properties than traditional centralized-manager/remote-agent workflow management systems.

## 3 PADRES System Description

The PADRES system is a distributed content-based p/s system which consists of a set of brokers connected by a peer-to-peer overlay network. Clients connect to brokers using various binding interfaces such as Java Remote Method Invocation (RMI) and Java Messaging Service (JMS). The PADRES subscription language is based on the traditional `[attribute,operator,value]` predicates used in several existing content-based p/s systems [3]. A subscription is a conjunction of predicates. Each message has a *mandatory* tuple describing the *class* of the message. The *class* attribute provides a guaranteed selective predicate similar to the topic in topic-based p/s systems. Consider a subscription `[class, eq, trigger] [appl, eq, payroll]`. This subscription matches all publications with class *trigger* where attribute *appl* is *payroll*.

### *3.1 Network Architecture*

The overlay network connecting the brokers is a set of connections that form the basis for message routing. The overlay routing data is stored in Overlay Routing Tables (ORT) on each broker. Specifically, each broker knows its neighbors from the ORT. Message routing in PADRES is based on the publish-subscribe-advertise model established by the SIENA project [3]. We assume that publications are the most common messages, and advertisements are the least common. A publisher issues an advertisement before it publishes. Advertisements are effectively flooded to all brokers along the overlay network using the ORT. A subscriber may subscribe at any time. The subscriptions are routed according to the Subscription Routing Table (SRT), which is built based on the advertisements in that broker. The SRT is essentially a list of `[advertisement,last hop]` tuples. If a subscription overlaps an advertisement in the SRT, it will be forwarded to the last hop of that advertisement. Subscriptions are routed hop by hop to the

advertiser, who may publish information of interest to the subscriber. Meanwhile, the subscription will be used to construct the Publication Routing Table (PRT). Like the SRT, the PRT is logically a list of `[subscription,last hop]` tuples, which is used to route publications. If a publication matches a subscription in the PRT, it will be forwarded to the last hop broker of that subscription until it reaches the subscriber. A diagram showing the overlay network, SRT and PRT is provided in Fig. 1. In this figure, an advertisement *1)* is propagated from $B_1$. A matching subscription *2)* enters from $B_2$, and is routed along the SRT. For instance, *2)* overlaps *1)* at broker $B_3$, and it is sent to $B_1$. Subscription *3)* does not overlap advertisement *1)*, so it is not forwarded by $B_4$. Lastly, publication *4)* matching subscription *2)* is routed along the PRT formed by *2)* to $B_2$. A subscription/advertisement covering and merging scheme [13, 15] is used to optimize content-based routing by reducing network traffic and routing table size, especially for applications with highly clustered data.

### *3.2 Broker Architecture*

The PADRES brokers are modular software components built on a set of queues: one input queue and multiple output queues. Each output queue represents a unique message destination. A diagram of the broker internals is provided in Fig. 2. The matching engine in the BrokerCore is built using the Java Expert System Shell (JESS) rule engine. It maintains the SRT and PRT, which are Rete trees [7]. For example, in the PRT, subscriptions are mapped to rules, and publications are mapped to facts, as shown in Fig. 3. The matching is performed by the JESS rule engine. When a new message is received by the broker, it will be put into the input queue. The matching engine takes the message from the input queue. If the message is a publication, it is inserted into the matching engine as a fact. When a publication matches a subscription in the PRT, its next hop destination is set to the last hop of the subscription, and it is placed into the corresponding output queue. If the message is a subscription, the matching engine first routes it according to the SRT, and, if there is an advertisement overlapping with the subscription, the subscription will be inserted into the PRT as a rule. Essentially, the rule engine performs matching and decides the next-hop destinations of the messages as a router. This novel approach allows for powerful subscription semantics and naturally enables composites subscriptions, which are more complex rules in the engine. Mapping the subscription language to a rule language is relatively straightforward. Extending this subscription language does not require significant changes in the engine. Furthermore, rule engines are well-studied, allowing PADRES to take advantage of existing research. The rule engine-based matching is quite efficient, especially for composite subscriptions.

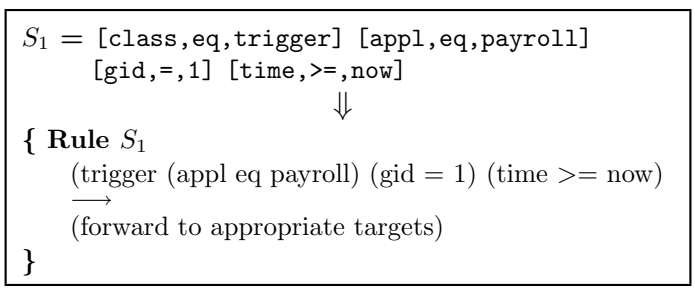


Figure 3: Mapping Subscription to Rule

### *3.3 Historic Data Access Description*

PADRES, unlike existing content-based p/s systems, allows the subscriber to subscribe to data published in both the future and the past. For future publications, PADRES uses the standard p/s messaging paradigm. Historic databases are attached to the brokers through a database binding. Databases store publications as they are published. Later, upon receiving a request for the historic data, the brokers re-publish relevant publications from their databases. Since, in content-based p/s, no direct addresses of participants are available, directly querying the databases for past data is impossible. Furthermore, it is not even possible for the client to know which database to query. Historical queries are routed to associated databases according to their content, and get the historic publications for clients. From the client's point of view, PADRES transparently delivers both past and future publications in the same manner (typically a Java callback). The powerful subscriptions supported in PADRES also allow the client to correlate past and future publications through temporal joins.

From the subscriber's perspective, the historic data access is performed by adding *time* predicates to standard subscriptions. If the specified *time* range includes some amount of time previous to the query time ("`now`"), the historic portion of the query will be split off and sent along the advertisement tree toward the appropriate data source(s). Since the historic databases advertise the content stored in them, the historic subscription will automatically be sent to the correct database(s). The databases, upon receiving the historic subscriptions, form an appropriate SQL query, retrieve the results from the data store, and re-publish the matching publications, which are routed to the subscriber along the subscription tree. Delivery of the historic publications happens in the same manner as standard (future) publications.

Simple historic queries are primitive subscriptions formatted as in a standard content-based p/s system. For example, `[class, eq, trigger][appl, eq, payroll][gid, =, $x] [time, <, now+1hr] [time, >, now-1hr]` matches all the trigger publications for application *payroll* published in a two hour window beginning one hour ago. Complex historic queries involve the use of composite subscriptions. These queries can be significantly more powerful. For instance, `[class, eq, job_status] [appl, eq, $y] [gid, =, $x] AND [class, eq, trigger], [appl, eq, $y] [gid, =, $x] [time, <, now]` matches the past triggers and job_status publications, which have the same generation ID and application name. The shared variables `$x` and `$y` cause the join. Perhaps the subscriber wants to monitor executions of all applications triggered in the past.

### *3.4 Composite Subscriptions*

**Subscription Language** The subscription language is used for subscribers to specify their interests. The language should provide several properties. First, the language should be powerful enough to describe complex subscriptions. It should support not only basic relational operators, such as `AND` and `OR`, but also advanced features, such as variables, *sequence* (`;`), and *repetition*. Second, the language should be notationally simple. Subscribers must be able to write composite subscriptions easily and succinctly. The syntax and semantics should be intuitive. Third, the language must have a mathematically precise concept of a match. When we specify a composite subscription, we know exactly which composite events will match it. Last, event patterns described by the language should be easy to detect. There is a trade-off between the language

expressiveness and detection efficiency. The more powerful the language is, the more complicated event pattern it can express, on the other hand, the more complex the composite event detection would be.

The subscription language of PADRES is augmented by the use of composite subscriptions. Each composite subscription is comprised of several component subscriptions with logical relational operators such as `AND` (∧), and `OR` (∨). Advanced operator *sequence* (;) represents the time sequence of two publications. For example, a publication matching $S_1$ is followed by a publication matching $S_2$. The event pattern satisfies the composite subscription $S_1;S_2$. A repetition event pattern is described as `Times(S, n, attr, v)`. It means publications matching $S$ happen $n$ times and attribute *attr* increases by step $v$; or decreases if $v$ is negative. These advanced operators are corresponding to key features supported in BPEL [10], such as structured activities *sequence* and *while*, etc. Variables are represented by $ in subscription predicates. `[class, eq, trigger] [appl, eq, $x]` ∧ `[class, eq, job_status] [appl, eq, $x]` is satisfied after publications of both *trigger* and *job_status* classes, of the same application, are published. These features are sufficient to support workflow management as described in Section 4.

A composite subscription is represented by a subscription tree, where the internal nodes are logical operators and leaf nodes are primitive subscriptions, as shown in Fig. 5. A composite subscription is mapped to a complex rule in the PRT, and each component subscription is part of the rule. When a publication matching one of the component subscriptions arrives, the composite subscription is stored in a partially matched state. When all components are matched, the composite subscription is considered matched. Composite subscriptions in PADRES can have variables bound to values in the publications.

**Composite Subscription Routing** If a composite subscription is matched, it means an event pattern called composite event is detected in the network. It is a higher level view of the publications. We perform a distributed composite event detection. The main difficulties when routing composite subscriptions are to decide the optimal placement to decompose them within the system and route the components to the correct destinations. With the knowledge of advertisements each subscription has its destination at a broker. Basically, if all component subscriptions have the same next hop destination, a broker would forward the composite subscription as a whole to the

**buildDestinationTree(*cs*):**
**Input:** Composite Subscription *cs*
**Output:** A destination tree *T*

```
1: Initialize T according to cs
2:     If (cs.root is leaf node)
3:         T.destination = cs.root.destination
4:     }else{
5:         T.left = buildDestinationTree(cs.root.left)
6:         T.right = buildDestinationTree(cs.root.right)
7:         If T.left.destination == T.right.desination
8:             T =T.left.destination
9:         }else{
10:            T =null
11:    }
12: Return T
```

Figure 4: Algorithm for Building a Destination Tree

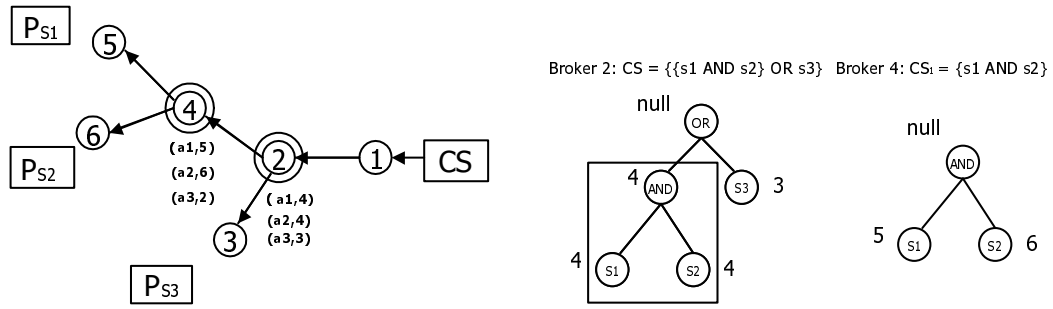


Figure 5: A Simple Composite Subscription Routing Example

destination; otherwise the composite subscription should be split into parts according to different destinations, and each part is forwarded to its own destination separately. The routing scheme is used to detect the event pattern matching a composite subscription at a location which is as close as possible to the data sources. A composite subscription is mapped to a single rule in the matching engine, which acts as both message router and event detector.

There are several advantages of using distributed composite event detection. Redundant detection is eliminated by sharing the detection results among subscribers. For the overlapping expressions of composite subscriptions issued by clients, the detection is executed once, and subscribers close to each other can reuse the detection results. Distributed detection also reduces network traffic. A composite subscription is forwarded into the network as far as possible before it is split. As a result, the number of subscriptions injected into the network does not increase rapidly because of the composite subscriptions. Furthermore, composite events are detected close to data sources in the network. A single notification is sent after a match, instead of a set of publications, reducing the number of publications routed in the federation.

When a broker receives a composite subscription, three routing steps are performed: first, a destination tree is built bottom-up for the composite subscription according to the SRT, which knows where all the component subscriptions come from. Leaf nodes are destinations of component subscriptions; an internal node is the destination of its child nodes if the two child nodes have the same destination, or *null* otherwise. If a node is *null*, all its parent nodes are *null*. The recursive algorithm for building such a tree is presented in Fig. 4. Second, the composite subscription is split according to its destination tree. Each node in the composite subscription tree has a corresponding node in the destination tree. If a node's destination is *null*, the subscription represented by the node is split into two parts, one for each child node. If a node's destination is not *null*, the node and its subtree will be routed as a single unit. The decomposition process is top-down. Third, all the split parts are routed to their destinations as specified by the destination tree.

Fig. 5 shows an example. Suppose a composite subscription `cs:` ($s_1$ `AND` $s_2$) `OR` $s_3$ comes from broker 1, and its matching publications come from different brokers. At broker 1, since all matching publications are coming from broker 2, `cs` is routed as a whole to broker 2. At broker 2, the SRT shows that publications matching $s_1$ and $s_2$ will come from broker 4, while $s_3$'s publications will come from broker 3. A destination tree is build, and `cs` is split into two parts: ($s_1$ `AND` $s_2$) and $s_3$. ($s_1$ `AND` $s_2$) is sent to broker 4, where it is split into $s_1$ and $s_2$, and sent to broker 5 and 6 separately. As a result, composite event pattern ($s_1$ `AND` $s_2$) is detected at broker 4 and the whole event pattern matching `cs` is detected at broker 2.

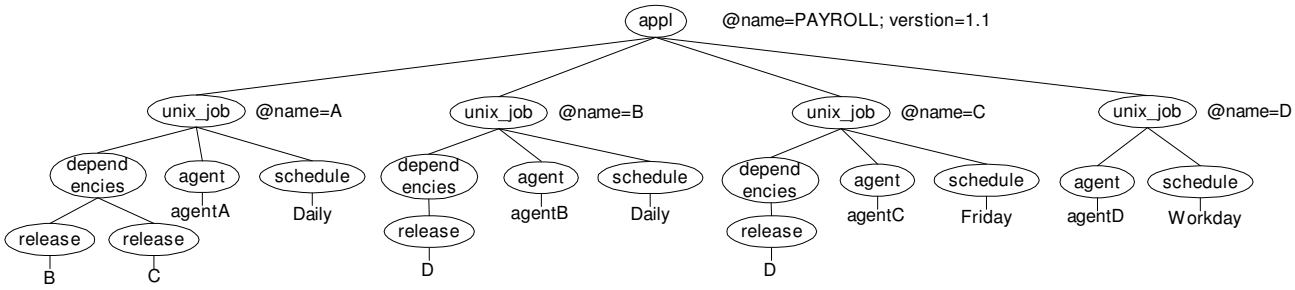


Figure 6: An XML Definition of a Workflow

## 4 Workflow Management

The p/s messaging paradigm has the potential to provide the foundation for new composite, Internet-scale, loosely coupled applications, such as workflow and business process management. PADRES, which has composite subscriptions and historic data access in addition to the standard p/s features, illustrates the success of the paradigm in the workflow management context. Our approach for workflow management discussed in the next section is based on business processes defined in XML. A business process described in BPEL can be converted to our XML schema and executed in PADRES.

### *4.1 Business-based Approach*

In traditional workflow management systems, jobs are grouped according to schedule time. Creating workflow definitions based on time means there may be multiple definitions for the same workflow. Consider, for example, there are four jobs in an application, as shown in Fig. 8. For this very simple workflow application, three execution pattern definitions exist, one each for *daily*, *workdays*, and *Fridays*. The advantage of this traditional time grouping based approach is that the workflow definitions are intuitive and easy to understand. However, the disadvantages are the following. First, it lacks scalability. It is an easy task to schedule when there are few jobs in an application, but it becomes much more difficult when managing thousands of jobs in one application, especially when the jobs are scheduled with a complicated combination of times. Second, one application may have several execution patterns based on time, and each pattern must have a definition. As a result, jobs have duplicate definitions in the patterns. When a workflow manager wants to modify one job, he has to modify all the copies of the job in each definition. This is expensive and prone to errors. Third, a large Enterprise Resource Planning (ERP) application always spans multiple platforms. This heterogeneity aggravates the traditional solutions by requiring more separate execution patterns. The cost of maintaining these multiple definitions is much higher than necessary.

Since a workflow is a set of business-related activities that are invoked in a specific sequence to achieve a business goal, it is intuitive to define a workflow from the business perspective. In the business-based approach, the emphasis is on the business process that needs to be accomplished rather than a more restrictive time-based definition. The business-based approach [4] for workflow management eliminates most of the problems of traditional time-based scheduling by reorganizing the way that jobs are defined. Fig. 6 demonstrates how the business-based approach works for the previous workflow example. This definition in XML can handle all three execution patterns with several advantages. First, each job is defined and stored only once. Duplicated job definitions

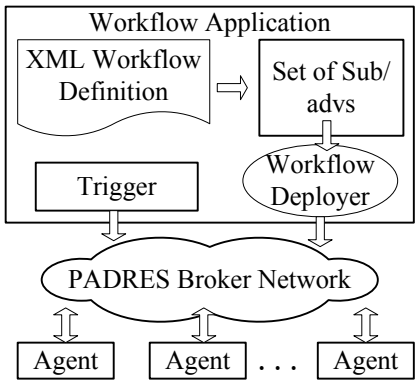


Figure 7: Workflow Management based on PADRES

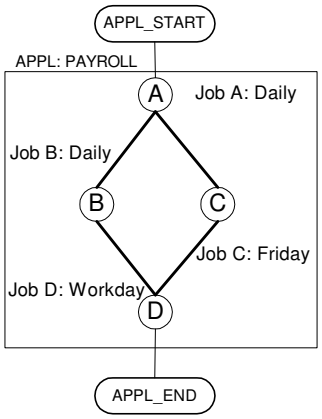


Figure 8: A Simple Application

are avoided. Second, a job can be executed in multiple periods. For instance, job `C` runs on every Friday. As a result, the whole application can run multiple generations. Third, scheduling in this manner allows for inheritable job dependencies. `D` depends on `B` and `C` if this application is executed on Friday. When the workflow runs on a workday other than Friday, `D` becomes a successor of `B` only. The XML workflow definition file identifies the jobs and describes the job dependencies in business process contexts. The XML document template could be easily extended to specify other dependencies, such as time and resources dependencies among jobs, as well as a more detailed job template including parameters and scripts that a job needs. Furthermore, different job templates can be defined for different jobs, such as database jobs.

### *4.2 Architecture Description*

A distributed p/s system, by providing a loosely-coupled messaging substrate, is by nature an ideal solution for workflow management. Since routing is content-based, the workflow manager does not need to maintain the address information of each job execution agent, which may be of significant size. The manager also does not bother to route the messages to and from agents, as those messages are automatically delivered using content-based routing. A workflow *manager* is a p/s client consisting of a workflow *deployer*, a *trigger issuer* and an *execution monitor*. Job execution agents are lightweight components without special logic for workflow management. They only need the capability to send and receive messages and execute jobs.

A workflow is deployed using publication and subscription messages, and then executed by assigned agents. There are several advantages of using a p/s system for workflow management. First, workflows are event-driven. A workflow is started by a trigger publication and is driven by publication messages of finished jobs. Control messages are automatically and transparently routed to the appropriate agents in the p/s layer. Second, workflows are easily scalable to multiple platforms, as the p/s architecture supports cross-platform applications in a distributed environment. Moreover, large-scale applications can be supported easily. Third, the management of workflow definitions is flexible. It is easy to add, modify or delete jobs from a workflow. The modification can be performed dynamically. Furthermore, job monitoring is a natural fit for the p/s paradigm, since managers can subscribe to job execution information. Fourth, multiple workflow applications can be deployed into the broker federation at the same time. The workflows of the applications can run concurrently without disturbing each other. Several generations of the same workflow can execute concurrently. Last, the distributed application deployment provides a robust workflow management

mechanism. Deploying a workflow application into a distributed network, instead of using a central manager to control the execution of the workflow, avoids a single point of failure.

Fig. 7 shows the architecture of a workflow management system based on PADRES. First, we need to transform an *XML workflow definition* file to a set of subscriptions and advertisements recognized by PADRES. Subscriptions describe the dependencies among jobs, i.e., the agent responsible for a job should subscribe to status messages for preceding jobs. Advertisements are used for agents to publish the execution information of a job when it is finished. Second, a workflow deployer uses *agent control messages* to dispatch these subscriptions and advertisements, and each agent will know how and when to run the assigned jobs. Third, *triggers* are used to start the execution of a workflow application. The format of the trigger is predefined. Once the trigger is issued, the first job of the workflow is executed. Only jobs whose schedule time match the trigger issue time can be run after the job dependencies are satisfied. After an agent finishes a job, it issues a *job_status* publication message indicating the success of the job, which continues to trigger its successors. The process continues until the whole workflow is finished. If a job has more than one predecessor job, it will not be executed until all the predecessor jobs are finished.

### *4.3 Workflow Transform*

Workflows are specified as XML documents detailing the order of job execution and the various dependencies between jobs. In order to run the workflow in PADRES, the XML documents are converted into a set of subscriptions and advertisements. There are two main steps in performing the transform. First, two virtual jobs are created (*APPL_START* and *APPL_END*), as shown in Fig. 8, which are used to simplify the transformation. Second, subscriptions and advertisements are generated for each job based on its predecessors and job information details. Fig. 9 illustrates the two steps.

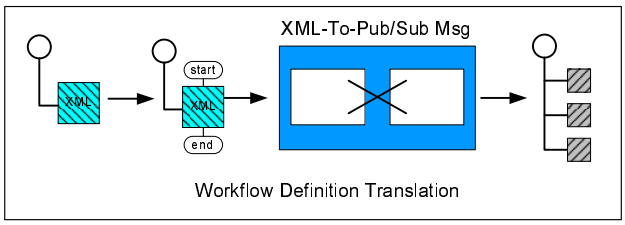


Figure 9: Workflow Definition Translation

There are two reasons to create the virtual jobs. First, the workflow chart of an application may be arbitrarily complicated, therefore the translation process needs to be generalized so that it can solve all the possible workflow definitions. In an application, the jobs that have no predecessors are called start jobs, for instance, job `A` is a start job in *payroll*. The jobs without successors are called end jobs, such as job `D`. *APPL_START* job is the predecessor of all the start jobs, and *APPL_END* job is the successor of all the end jobs. No matter how complex the application is, it starts from *APPL_START* and ends in *APPL_END*. Each job in the definition can be processed identically without distinguishing start and end jobs from other jobs. The second reason is that it is hard to tell when the application is started and when it is finished because of the multiple start jobs and end jobs. The start and end states of the whole application are indicated

by the two virtual jobs, so the monitor can determine workflow execution status at the application level.

Three kinds of p/s messages are created from the XML definition file for each job in the workflow: job dependency subscriptions, strings of job execution information, and advertisements for *job_status* messages. The job dependency subscription is a composite subscription, as a job depends not only on all its predecessor jobs, but also on trigger instances. Start jobs in an application subscribe to job *APPL_START*, and job *APPL_END* subscribes to all end jobs. Other jobs subscribe to their own predecessors. Every job subscribe to triggers as well. A job information string contains detailed information about the job execution, such as when the job is scheduled and what parameters the job needs. Advertisements corresponding to the job information publications are created since a publisher must advertise before publishing.

### *4.4 Workflow Deployment*

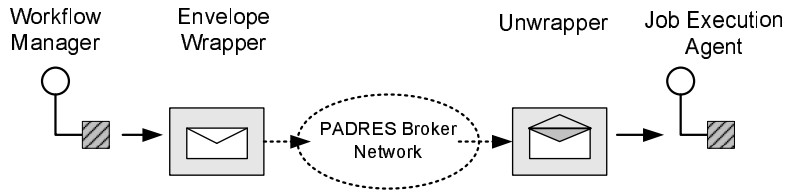


Figure 10: Envelope Wrapper

The goal of workflow deployment is to send the subscriptions and advertisements generated from the workflow definition file to the corresponding *job execution agents*. These messages related to particular jobs in the workflow should be issued by proper job execution agents. In the p/s paradigm, the workflow manager uses an *envelope wrapper* to wrap the raw messages inside envelopes that are compliant with the p/s messaging infrastructure. An agent control publication of class *agent_ctl* is the envelope. The agent control messages carry the information that the manager wants to deliver to a particular agent, indicating the agent name in the *agent* predicate. Agents receive the control messages by subscribing as usual. The p/s broker network delivers the *agent_ctl* publications to the proper agents. After the agents receive the envelopes, they unwrap them, extract the subscriptions/advertisements from the envelopes, and issue the messages as their own subscriptions/advertisements, as shown in Fig. 10. As a result, these agents are both subscribers and publishers. They subscribe to *job_status* messages for predecessor jobs and publish *job_status* messages for their own jobs.

The p/s architecture makes it easy to maintain the workflow definition in the distributed broker network. For example, if a job definition needs to be modified, the manager can use *agent_ctl* messages to ask the corresponding agent to change the job information table. If a job is to be deleted, the manager can ask the agent to unsubscribe the subscription for that job. In this case, the subscriptions of its immediate successors should be modified as well.

### *4.5 Workflow Execution*

Job execution agents are both subscribers and publishers. The dual roles enable them to exchange messages within the p/s messaging system, allowing coordinated execution of the workflow. Each execution instance of an application is a generation of the workflow.

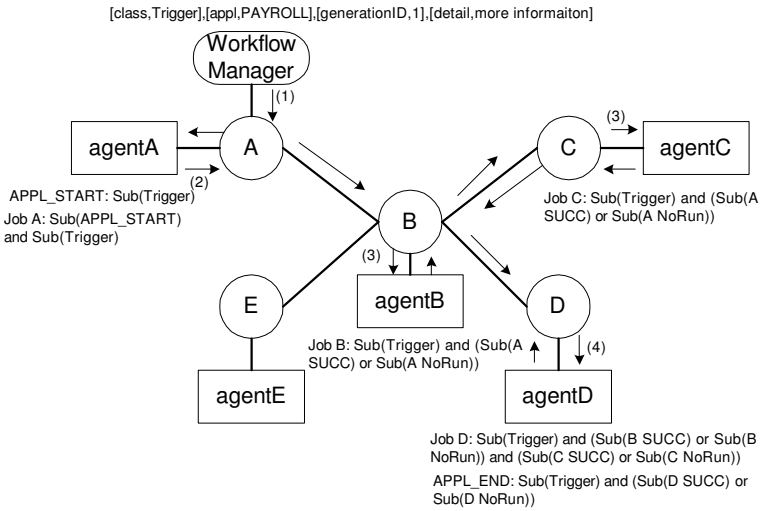


Figure 11: Workflow Execution

Unique *generation IDs* are used to distinguish between different generations of the same application. A particular generation of an application is started by a *trigger* publication. `APPL_START` is executed first. It fires all start jobs. The workflow execution is driven by finished jobs. Execution continues until all the jobs defined in the application are finished. `APPL_END` is triggered by the end jobs. An example is given in Fig. 11 to illustrate the process of workflow execution. The key to workflow execution is job dependency subscriptions, which determine the order of execution for the jobs. All the message routing is automatic and transparent to the workflow management layer. Triggers issued at different times will result different execution patterns because of the dynamic and inheritable job dependencies.

A series of messages of a generation has the same *generation ID*. After the generation is over, these out-of-date messages are removed from the brokers and agents. As a result, the brokers can perform later executions efficiently. Moreover, multiple generations of an application can run concurrently. Since a workflow may have a long time span of execution, concurrent executions are necessary and efficient. Similarly, several different applications can be deployed in the broker network. Workflows of different applications can run concurrently.

### *4.6 Workflow Monitoring*

A workflow management system maintains a trace of job executions and provides a control and monitoring interface. One of the key functions of a workflow management system is job monitoring, which has two levels: realtime monitoring and historic query. Realtime monitoring is easy to fit in the content-based p/s paradigm. The monitor could subscribe to the job information publications of a particular job. As a result, when the job is finished, the monitor would know the execution status information. The monitor could monitor the execution at the job and application levels. With the historic data access supported by PADRES, the workflow management system can answer queries for previously executed workflows. Databases store workflow publications as they are published. The monitor may retrieve these publications by issuing historic queries. Moreover, with the powerful composite subscriptions, the monitor can monitor both the executing and previously executed workflows at the same time. PADRES also provides a graphical interface which allows the monitor to visualize the network topology and message routing in order to have a intuitive picture of the workflow execution.

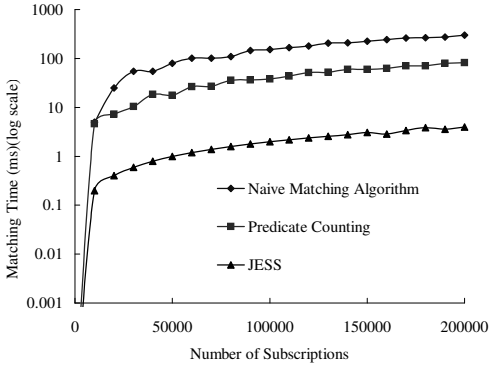


Figure 12: Matching Time

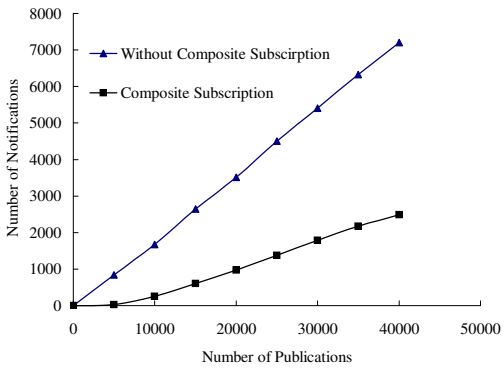


Figure 13: Notifications

## 5 Evaluation

**Evaluation Setup** We implement PADRES in Java with JDK1.4.2 using the Java Expert System Shell (Jess) [8] as a matching engine and use RMI as the native transport protocol. All our experiments are performed on a computer with an Intel Xeon 3GHz processor and 2GB RAM, of which 1GB is allocated to the JVM. For the experiments of the distributed workflow management system, we deployed a simple distributed network with 5 overlay brokers, and suppose equal latencies between brokers.

**Matching Time** To evaluate the performance of the rule-based matching engine, we generated 200,000 subscriptions and 50,000 publications. Lack of the benchmarks and real application data, the subscriptions and publications were generated by a workload generator which produces the data randomly by selecting attributes from a list of twenty attributes and selecting values from given value ranges. We assume that the value of each attribute in a publication is uniformly selected from its value range.

Fig. 12 shows the matching time of publications against subscriptions. The matching time is given using a logarithmic scale. Each data point is obtained by averaging the time taken to route 5,000 publications. We compare the rule-based matching engine implemented in Jess with two other methods. One is a naive matching algorithm which linearly scans the routing table to find the matched subscriptions. The other is a matching algorithm that is similar to the predicate counting algorithm [1]. This algorithm calculates distinct predicates only once. Our experiments show that the rule-based matching engine is much faster that the two methods. The Rete algorithm used in Jess is very efficient [7]. It takes only $4.52ms$ to route a publication against 200,000 subscriptions. This excellent matching efficiency indicates that our engine is suitable for large scale p/s systems and can process a large number of publication and subscription messages efficiently.

**Network Traffic Overhead** Detecting composite events in the broker network reduces the message traffic received by clients. To illustrate this, we compare two scenarios. In the first scenario, a client issues 200 composite subscriptions, each consisting of 5 primitive subscriptions. In the second scenario, no composite subscriptions are supported, so the client issues the 1000 primitive subscriptions that made up the original 200 composite subscriptions. After a large number of publications are injected into the broker network, we measure the number of notifications received by the client in the different scenarios, as shown in Fig. 13. The result shows that the number of notifications sent to the client is greatly reduced by the composite subscriptions.

We also measure the network traffic overhead of the workflow deployment and execution. We design two workflows: workflow $A$ is an application with four jobs. Workflow $B$ is an application with eight jobs, which is more complex than $A$. The manager issues *agent_ctl* messages to dispatch the workflow to agents, and agents send out subscription

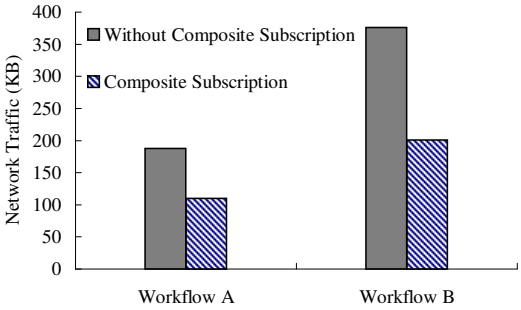


Figure 14: Deployment Traffic

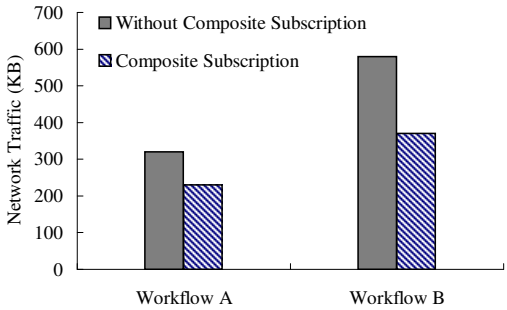


Figure 15: Execution Traffic

and advertisement messages, which indicate the job dependencies. Without composite subscriptions, agents have to subscribe to several primitive subscriptions instead of a composite one, so more messages must be sent in the broker network. Meanwhile, agents need special logic to check whether a job running condition is satisfied. Supposing each publication and subscription message is 1KB, and the cost of sending a message from data source to data sink is the number of hops, we measure the traffic overhead of the deployment in Fig. 14. The results show that more messages are deployed if there is no composite subscription function provided. Composite subscriptions reduce the network bandwidth by about 55% in both workflow. Workflow $B$ is more complex than $A$, so the traffic overhead of $B$ is also larger than $A$.

The traffic overhead caused by ten generations of workflow executions is shown in Fig. 15. The composite subscription function can save network bandwidth and distributed composite event detection can further reduce the overhead, which is reduced to 62.7% in workflow $B$. The execution process is started by an issued trigger instance. When a composite subscription issued by an agent is matched, only one notification message is sent back to the agent, as opposed to several individual primitive notifications. For example, if a job depends on ten other jobs, instead of forwarding ten messages to the agent, only one notification message is forwarded to the agent in order to decide whether to run the job or not. As a result, agents do not need particular logic to process the received messages, as composite event detection is performed in the broker network.

The network traffic overhead is $O(n)$, where $n$ is the size of a workflow, that is the number of jobs involved in the workflow. The more complicated the workflow is, the more benefit we can gain from composite subscriptions and distributed composite event detections, both in the deployment and the execution phases.

## 6 Conclusions

In this paper we have presented PADRES, a distributed content-based p/s system built for the requirements of workflow management. PADRES was developed in collaboration with Cybermation, Inc., developer of the ESP Workload Manager and ESP Espresso products. The p/s brokers use a rule-based matching engine to perform content-based message matching and routing in p/s. PADRES provides a subscription language so that subscribers can express their complex interests as composite subscriptions. The novel historic data access module supplies a hybridized means for subscribers to retrieve both past and future information from a p/s system. A next-generation distributed workflow management system based on PADRES is presented. Our experiments show that the rule-based matching engine based on a Rete works efficiently, and the distributed workflow management system successfully performs job execution and monitoring with the help of composite subscriptions and historic data access.

## References


[1] G. Ashayer, H. Leung, and H.-A. Jacobsen. Predicate matching and subscription matching in publish/subscribe systems. In *DEBS'02 Workshop at ICDCS'02*, Vienna, Austria, 2002.

[2] I. Burcea, H.-A. Jacobsen, E. de Lara, V. Muthusamy, and M. Petrovic. Disconnected Operation in Publish/Subscribe Middleware. *2004 IEEE International Conference on Mobile Data Management.*

[3] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.

[4] Cybermation, Inc. Corporate website. `http://www.cybermation.com`.

[5] U. Dayal, M. Hsu, and R. Ladin. Organizing long-running activities with triggers and transactions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 204–214, 1990.

[6] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[7] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[8] E. J. Friedman-Hill. Jess, The Rule Engine for the Java Platform. `http://herzberg.ca.sandia.gov/jess/`.

[9] C. Hagen and G. Alonso. Flexible exception handling in the OPERA process support system. In *Proceedings of the The 18th International Conference on Distributed Computing Systems*, page 526. IEEE Computer Society, 1998.

[10] IBM and Microsoft. Business process execution language for web services version 1.0. `http://dev2dev.bea.com/techtrack/BPEL4WS.jsp`.

[11] Informix Software. Inormix Media360. `http://www.informix.com/media360`, 2000.

[12] JBoss, Inc. Corporate website. `http://www.jboss.com/`.

[13] G. Li, S. Hou, and H.-A. Jacobsen. A unified approach to routing, covering and merging in publish/subscribe systems based on modified binary decision diagrams. *International Conference on Distributed Computing Systems (ICDCS'05), Columbus, Ohio, USA*, 2005.

[14] H. Liu and H.-A. Jacobsen. Modeling uncertainties in Publish/Subscribe System. In *In Proceedings of ICDE*, 2004.

[15] G. Mühl. Generic constraints for content-based publish/subscribe systems. In C. Batini, F. Giunchiglia, P. Giorgini, and M. Mecella, editors, *Proceedings of the 6th International Conference on Cooperative Information Systems (CoopIS '01)*, volume 2172 of *LNCS*, pages 211–225, Trento, Italy, 2001. Springer-Verlag.

[16] L. Opyrchal, M. Astley, J. Auerbach, G. Banavar, R. Strom, and D. Sturman. Exploiting IP multicast in content-based publish-subscribe systems. In *IFIP/ACM International Conference on Distributed systems platforms*, pages 185–207. Springer-Verlag New York, Inc., 2000.

[17] Oracle Corporation. Oracle Workflow Technical configuration in Oracle Applications Release 11. `http://www.oracle.com/support/`, 2000.

[18] M. Petrovic, I. Burcea, and H.-A. Jacobsen. S-ToPSS - a semantic publish/subscribe system. In *Very Large Databases (VLDB'03)*, Berlin, Germany, September 2003.

[19] M. Petrovic, H. Liu, and H.-A. Jacobsen. G-ToPSS - fast filtering of graph-based metadata. In *the 14th International World Wide Web Conference (WWW2005*, Chiba, Japan, May 2005.

[20] P. R. Pietzuch and J. Bacon. Hermes: A distributed event-based middleware architecture. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 611–618. IEEE Computer Society, 2002.

[21] P. R. Pietzuch, B. Shand, and J. Bacon. Composite event detection as a generic middleware extension. *IEEE Network Magazine, Special Issue on Middleware Technologies for Future Communication Networks*, January/February 2004.

[22] Z. Xu and H.-A. Jacobsen. Efficient constraint processing for location-aware computing. In *6th International Conference on Mobile Data Management (MDM'05), Ayia Napa, Cyprus*, 2005.