A Distributed Service Oriented Architecture for Business Process Execution

GUOLI LI, VINOD MUTHUSAMY, and HANS-ARNO JACOBSEN
University of Toronto

The Business Process Execution Language (BPEL) standardizes the development of composite enterprise applications that make use of software components exposed as Web services. BPEL processes are currently executed by a centralized orchestration engine, in which issues such as scalability, platform heterogeneity, and division across administrative domains can be difficult to manage. We propose a distributed agent-based orchestration engine in which several light-weight agents execute a portion of the original business process and collaborate in order to execute the complete process. The complete set of standard BPEL activities are supported, and the transformations of several BPEL activities to the agent-based architecture are described. Evaluations of an implementation of this architecture demonstrate that agent-based execution scales better than a non-distributed approach, with at least 70% and 120% improvements in process execution time, and throughput, respectively, even with a large number of concurrent process instances. In addition, the distributed architecture successfully executes large processes that are shown to be infeasible to execute with a non-distributed engine.

Categories and Subject Descriptors: H.4 [**Information Systems Applications**]: Miscellaneous

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: business process, BPEL, workflow management, service-oriented architecture, distributed orchestration, publish/subscribe

1. INTRODUCTION

Enterprise applications are increasingly being architected in a service-oriented architecture (SOA) style, in which modular components are composed to implement the business logic. The properties of such applications, such as the loose coupling among the modules, is promoted as a way for an agile business to quickly adapt its processes to an ever changing landscape of opportunities, priorities, partners, and competitors. The proliferation of Web services standards in this area reflects the industry interest and demand for distributed enterprise applications that communicate with software services provided by vendors, clients, and partners.

For example, an online retailer may utilize the services of a partner shipping company to allow their customers to track the delivery status of products. The shipping company here would expose a component that allows its partners to retrieve delivery status information. Other external services the retailer may use include a

Author's address: Hans-Arno Jacobsen, Dept. of ECE and Dept. of CS, University of Toronto, 10 King's College Road, Toronto, Ontario, Canada, M5S 3G4; email: jacobsen@eecg.toronto.edu

payment service (such as PayPal), or a product review database. In addition, the retailer may use services developed internally, such as a user interface engine (to render interfaces for various devices such as a PDA or PC), and an authentication service. Developing such loosely coupled components makes it easier to develop, maintain, and modify the application.

It is not uncommon for business processes in industries such as supply chain management, online retail, or health care to consist of complex interactions among a large set of geographically distributed services developed and maintained by various organizations. The processes themselves can be very large, long running, manipulate vast quantities of data, and require thousands or millions of concurrent process instances. For example, one of our project partners reports that a large Chinese electronics manufacturer employs formal business processes to drive its operations activities including component stocking, manufacturing, warehouse management, order management, and sales forecasting. The processes are inherently distributed using department-level processes for manufacturing, warehouse and order management, with each of these processes utilizing from 26 to 47 activities. There also exist global processes that compose the department-level ones. In addition to the separation by administrative domains, the processes also involve geographically distributed parties including a number of suppliers, several organizational departments, a dozen sales centers, and many retailers. Thousands of instances of these processes are executing concurrently at any one time. Such large processes involving dozens of collaborating parties is a natural fit for our distributed execution architecture.

Business processes are executed by an *orchestration engine* that is responsible for carrying out the activities in the process, and maintaining the state associated with process instances. Typically a single engine is deployed to manage an application, and scalability is addressed by replicating the engine. Many existing BPEL engines support clustering in order to optimize and ensure business process throughput on highly available systems. When a business process needs to be scaled to meet heavier processing needs, the BPEL engine's clustering algorithm automatically distributes processing across multiple engines. In this paper, we present NIÑOS, a completely different orchestration architecture, one that is more in agreement with the distributed nature of the processes themselves.

The NIÑOS runtime orchestrates business processes by distributing process execution across several light-weight agents, each of which carry out a single activity. This distributed architecture is congruent with an inherently distributed enterprise where business processes are geographically dispersed and coordinating partners have to communicate across administrative domains. Not only does NIÑOS remove the scalability bottleneck of a centralized orchestration engine, it offers additional efficiencies by allowing portions of processes to be executed close to the data they operate on, thereby conserving data and control traffic. Furthermore, NIÑOS supports flexible mappings of the orchestration agents onto heterogeneous platforms and resources, permitting the system to shape itself from a centralized to a fully distributed configuration.

NIÑOS utilizes and exploits the rich SOA enterprise service bus (ESB) capabilities of the PADRES distributed content-based publish/subscribe routing infrastructure [Fidler et al. 2005; Li and Jacobsen 2005; Li et al. 2008; Hu et al. 2009; Kazemzadeh and Jacobsen 2009]. All communication in the system occurs as pub/sub interactions, including process coordination among the agents, control and monitoring. This decouples the agents, which now only need to be aware of one another's *content-based* addresses, thereby simplifying agent reconfiguration and movement, and seamlessly allowing multiple processes and process instances to coexist. In addition, in NIÑOS, processes are transformed such that certain computations are carried out in the pub/sub layer, exploiting advanced features available in PADRES. This further simplifies the orchestration agents, and allows these computations to be optimized by the PADRES layer by, for example, performing in-network event correlation. Yet another advantage afforded by the pub/sub layer is ease of administration. Agents can be configured and controlled individually, or as some subset, using their location-independent content-based addresses. Similarly, since all communication occurs over the pub/sub layer, the system can be fully monitored without additional instrumentation logic. The declarative pub/sub interface supports expressive queries for precisely the information of interest.

The contributions of this paper include, (1) the design of the NIÑOS distributed business process execution architecture based on the flexible PADRES pub/sub layer; (2) a procedure to map standard Business Process Execution Language (BPEL) processes, including the complete set of BPEL activities, to a set of distributed NIÑOS agents, with control flow realized using decoupled pub/sub semantics; and (3) an evaluation of the NIÑOS orchestration engine that demonstrates its improved scalability over a centralized engine.

We present background and related work in Section 2, followed by a description of the BPEL mapping process to the NIÑOS system architecture in Section 3, an evaluation of NIÑOS in Section 4, and some concluding remarks in Section 5.

2. BACKGROUND AND RELATED WORK

In order to keep the paper self-contained, this section presents a brief overview of the BPEL language and the pub/sub model, and places NIÑOS within the context of related work.

BPEL: The Business Process Execution Language (BPEL) standard supports writing distributed applications by composing, or *orchestrating*, Web services. A BPEL process consists of a set of predefined *activities*. BPEL programs have properties of traditional programming languages (with concepts of scope, variables, and loops) and workflows (with concepts of parallel and sequential flows). BPEL processes are often authored in a proprietary graphical tool that serializes the process into a standard BPEL XML file.

BPEL activities can be classified as *basic* activities that perform some primitive operation such as receiving a message or throwing an exception, and *structured* activities that define control flow. The key BPEL activities are summarized in Table I.

Several vendors have implemented BPEL engines, including IBM, Microsoft, Oracle, and Sun Microsystems. Scalability is typically addressed by load balancing process instances across a cluster of engines, where each engine still executes the entire process. In NIÑOS, however, the individual activities within a process are

Basic Activities

Activity	Description
receive	Blocking wait for a message to arrive.
reply	Respond to a synchronous operation.
assign	Manipulate state variables.
invoke	Synchronous or asynch. Web service call.
wait	Delay execution for a duration or deadline.
throw	Indicate a fault or exception.
compensate	Handle a fault or exception.
terminate	Terminate a process instance.

Structured Activities

Activity	Description
sequence	Sequential execution of a set of activities.
while	Looping constructs.
switch	Conditional exec. based on instance state.
pick	Conditional exec. based on events.
flow	Concurrent execution.

Table I. Summary of BPEL activities.

distributed among the available computing resources. The latter design also allows placing computational activities near the data they operate on, which is not possible in the cluster architecture. Furthermore, NIÑOS is applicable to the realization of cross-enterprise business process management, where no one single entity runs and controls the entire business process, but rather the process emerges as a choreographed concert of activities and sub-processes run by each organization.

Publish/Subscribe: In pub/sub communication, the interaction between the information producer (*publisher*) and consumer (*subscriber*) is mediated by a set of *brokers* [Fabret et al. 2001; Carzaniga et al. 2001]. Publishers *publish events* to the broker network, and subscribers *subscribe* to interesting events by submitting *subscriptions* to the broker network. It is the responsibility of the brokers to route each event to interested subscribers. In *content-based* pub/sub, subscribers can specify constraints on the content of the events, and the broker network is said to perform *content-based routing* of events. The terms *event* and *publication* are often used synonymously in the pub/sub literature and in this paper.

The routing in our PADRES distributed content-based pub/sub system works as follows [Fidler et al. 2005]. Publishers and subscribers connect to one of the brokers in a broker overlay network. Publishers specify a template of their event space by submitting an *advertisement* message that is flooded through the broker network and creates a spanning tree rooted at the publisher. Similarly, subscribers specify their interest by sending a *subscription* message that is forwarded along the reverse paths of *intersecting* advertisements, i.e., those with potentially interesting events. Now *publications* from publishers are forwarded along the reverse paths of *matching* subscriptions to interested subscribers.

PADRES extends traditional pub/sub semantics with *composite subscriptions* that allow event correlations to be specified [Jacobsen ; Li and Jacobsen 2005]. For example, a subscriber may only be interested in being notified of business processes with at least two failed instances within an hour. The correlation computations are performed at strategic points in the broker network. Another PADRES extension is *historic queries* [Fidler et al. 2005; Li et al. 2007] which allow subscriptions to query for events published in the past in addition to those in the future. This is useful in enterprise applications where past system events may be needed for auditing or analysis purposes.

Distributed workflows: Distributed workflow processing has been studied in the 1990s to also address scalability, fault resilience, and enterprise-wide workflow management [Alonso et al. 1995; Wodtke et al. 1996; Muth et al. 1998]. Alonso *et al.* present a detailed design of a distributed workflow management system [Alonso et al. 1995]. The work bares similarity with our approach in that a business process is fully distributed among a set of nodes. However, the distribution architectures differ fundamentally. In our approach, a content-based message routing substrate is built to naturally enable task decoupling, dynamic reconfiguration, system monitoring, and run-time control. This is not addressed in the earlier work. Moreover, we present a proof-of-concept implementation and detailed performance results comparing distributed and centralized workflow management architectures, which is lacking in prior approaches.

A behavior preserving transformation of a centralized activity chart, representing a workflow, into an equivalent partitioned one is described in by Muth *et al.* [1998] and realized in the MENTOR system [Wodtke et al. 1996]. The objective of the work is to enable the parallel execution of the partitioned flow, while minimizing synchronization messages, and to analytically prove certain properties of the partitioned flow [Muth et al. 1998]. In a different set of transformations, parallelizing compiler-inspired techniques, including control flow and data flow analysis, are used to parallelize the business process to achieve the highest possible concurrency [Nanda et al. 2004]. Both the transformation papers are complementary to our work since we operate with the original business process model without analyzing the process. An advantage of executing an unmodified process is that dynamic changes to the executing business process instances are possible, as their structure remains unchanged from the original specification.

Casati *et al.* present an approach to integrate existing business processes within a larger workflow [Casati and Discenza 2001]. They define event points in business processes where events can be received or sent. Events are filtered, correlated, and dispatched using a centralized publish/subscribe model. The interaction of existing business processes is synchronized by event communication. This is similar to our work in terms of allowing business processes to publish and subscribe. In our approach, activities in a business process are decoupled and are executed by activity agents, which are publish/subscribe clients, and the communication between agents is performed in a content-based publish/subscribe broker network.

Stream processing: There has been a lot of work on distributed stream processing engines in which a set of *operators* are installed in the network to process streams of data and execute SQL-like queries over the data streams [Kumar et al. 2006; Abadi et al. 2005; Chandrasekaran et al. 2003; Pietzuch et al. 2006]. These operators input and output a set of streams and may filter, or change the data on these streams.

Borealis is a distributed stream processing engine in which streams are queried by a network of operators [Abadi et al. 2005]. In addition to using a proprietary query language, Borealis does not support loops in the query network, which makes it unsuitable for general business process execution, specified in BPEL or similar languages, where looping constructs are commonplace.

In the IFLOW distributed stream processing engine, IFLOW nodes are organized in a cluster hierarchy, with nodes higher in the hierarchy assigned more responsibility [Kumar et al. 2006]. For example, the root node is responsible for deploying the entire operator network to its children, and for monitoring the summarized execution statistics of this network. This is different from our completely distributed architecture in which brokers have equal responsibility.

While stream processing engines may bear some architectural resemblance to a set of agents executing a business process, there are issues related to business process execution that are not easily handled by stream processing engines. First, the stream processing work above is based on proprietary languages, not an industry standard such as BPEL. More significantly, a business process is conceptually not simply a data stream. There are notions of process instances and the accompanying state and isolation semantics that are not required in streams.

In addition to the semantic differences between processes and streams, process distribution in NIÑOS differs from the above work by exploiting an underlying content-based pub/sub system. As in IFLOW, our agents are decoupled by communicating using pub/sub content-based addresses instead of network identifiers. In addition, we utilize the *composite subscription* feature in PADRES to offload some of the agent processing to the pub/sub network. This simplifies the agents, and allows the pub/sub network to optimize this processing logic. This is further explained in Section 3.

Dynamic redeployment: While this paper demonstrates the benefits of a distributed process execution architecture, it leaves open the problem of how a process should be deployed in order to satisfy certain goals. Ongoing work in this area attempts to dynamically redeploy the distributed business process based on certain the current workload, environment, and process goals expressed as formal service level agreements [Chau et al. 2008; Muthusamy et al. 2009]. It is important in this situation that the behavior of the process is unaffected by the redeployment of certain components. To support this capability, prior work has developed the notion of a transactional movement operation in which the redeployment of stateful pub/sub clients is guaranteed to satisfy a number of formal properties [Hu et al. 2009].

This paper builds on our extensive prior work on building the PADRES system, a scalable, distributed pub/sub messaging middleware. The NIÑOS architecture exploits capabilities of the PADRES system including fine-grained content-based routing [Fidler et al. 2005], and support for event correlations using composite subscriptions [Li and Jacobsen 2005]. Other notable features of the PADRES system, such as user-tunable fault-tolerance [Kazemzadeh and Jacobsen 2009], load-balancing [Yeung Cheung and Jacobsen 2006], system policy management [Wun and Jacobsen 2007], historic data access [Li et al. 2007], and the ability to route in cyclic overlay networks [Li et al. 2008] are useful infrastructural properties for mission-critical enterprise applications.

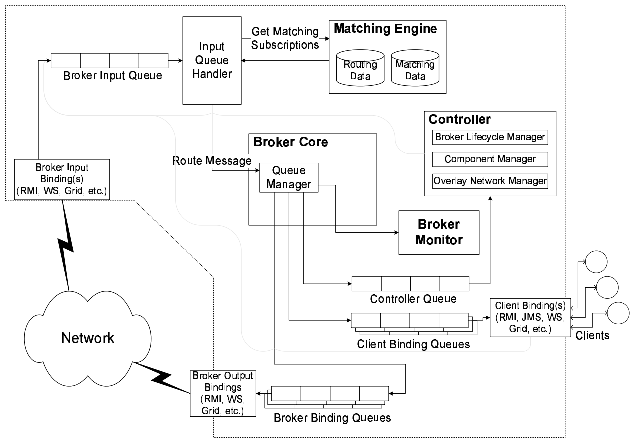


Fig. 1. PADRES broker architecture

3. DISTRIBUTED PROCESS EXECUTION

NIÑOS is a distributed business process execution architecture. It leverages the PADRES pub/sub system by transforming a BPEL business process into fine-grained pub/sub agents that collaborate to realize the original process. These agents interact using pub/sub messages and take advantage of some of the in-network processing capabilities available in PADRES. To simplify management, NIÑOS allows processes to be deployed and monitored in a centralized manner, again exploiting some of the decoupling properties of the PADRES pub/sub system.

3.1 PADRES system description

The PADRES system is a distributed content-based pub/sub system which consists of an overlay network of brokers, where clients connect to brokers using Java Remote Method Invocation (RMI) or Java Messaging Service (JMS) interfaces.

Network architecture: The overlay network connecting the brokers is a set of connections that form the basis for message routing, with each broker knowing only about its immediate neighbors. Message routing in PADRES is based on the pub/sub content-based routing model, where publications are routed toward interested subscribers who have expressed interest in receiving publication content by issuing subscriptions. At the application level, only publish and subscribe primitives are available and no IP address information is required. All distributed client interactions take place in this manner. A subscriber may subscribe at any time, and publications are exclusively disseminated to subscribers who have issued matching subscriptions.

Broker architecture: The PADRES brokers are modular software components built on a set of queues: one input queue and multiple output queues, with each output queue representing a unique message destination. A diagram of the broker internals is provided in Figure 1. The matching engine, a critical component of a broker, maintains various data structures. In one such structure subscriptions are mapped to rules, and publications are mapped to facts. The rule engine performs

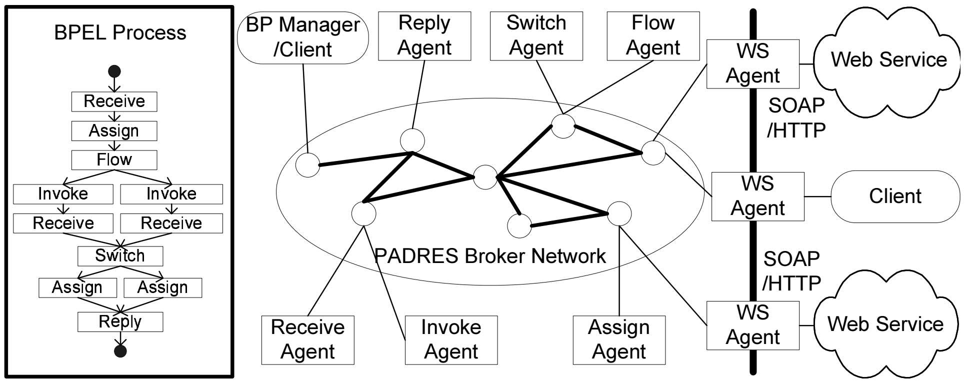


Fig. 2. NIÑOS distributed business process execution architecture

matching and decides the next-hop destinations of the messages. This novel rule-based routing approach allows for powerful subscription semantics and naturally enables composites subscriptions, which are more complex rules in the rule engine. Mapping the subscription language to a rule language is relatively straightforward, and 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. Our experience with the system indicates that rule-based matching is quite efficient, especially for composite subscriptions.

3.2 NIÑOS system architecture

The NIÑOS system architecture, as shown in Figure 2, consists of four components: the underlying PADRES broker network, activity agents, Web service agents, and a business process manager. As mentioned in Section 2, the PADRES *broker network* consists of a network of brokers that carry out content-based routing and in-network processing of composite subscriptions.

In NIÑOS, each business process element, such as a BPEL activity, has a corresponding *activity agent*, which is a light-weight pub/sub client. Generally, an agent waits for its predecessor activities to complete by subscribing to such an event, then executes its activity, and finally triggers the successor activities by publishing a completion event. As a result, process execution is event-driven and naturally distributed.

Cross-enterprise business interaction is a requirement in business processes. For example, BPEL supports invoking partner Web services. NIÑOS *Web service agents* interface Web services with the PADRES network by translating between Web service protocols (such as SOAP over HTTP) and pub/sub message formats. This allows the appropriate activities in a NIÑOS business process to invoke and be invoked by external Web services. Web service agents support both static partners, which are defined at design time, and dynamic partners, determined at runtime.

The *business process manager*, which is also a pub/sub client, transforms business processes into pub/sub messages for the activity agents, deploys the process onto the available agents in the network, triggers instances of business processes, and monitors and controls the execution.

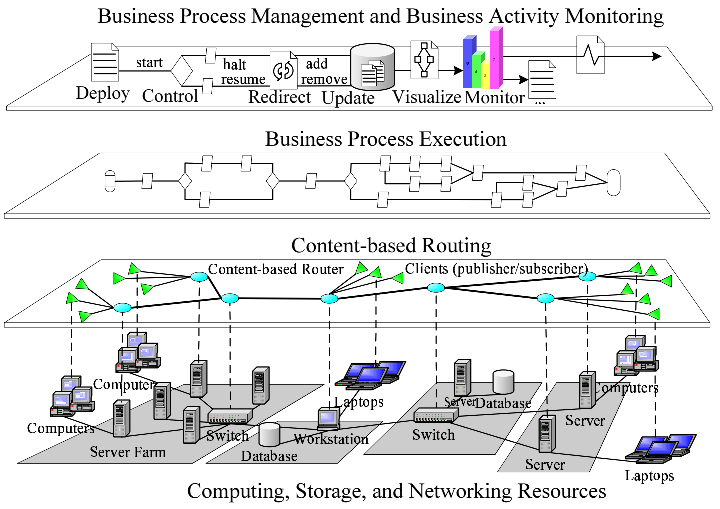


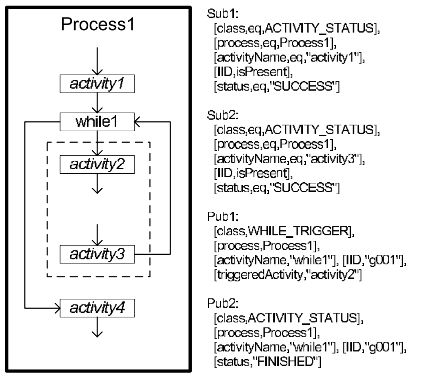
Fig. 3. Conceptual system architecture

NIÑOS addresses three phases of business process execution: process transformation, deployment, and execution. In the *transformation* phase, a business process is mapped to a set of activity agents and corresponding pub/sub messages that specify the dependencies among the activities. The transformation of some interesting BPEL activities is described in Section 3.3 in detail.

In the *deployment* phase, the business process manager deploys the process to the appropriate activity agents. Each activity agent subscribes to *agent control messages* with a unique agent identifier, allowing the manager to install an activity at a particular agent. An agent partakes in a business process by issuing the subscriptions and advertisements as requested by the manager, thereby building up the inter-agent activity dependencies and making the process ready to execute.

In the *execution* phase, the deployed business process can be invoked through a Web service agent, which translates the invocation into a NIÑOS service request. The service request is a publication message that specifies the process and instance identifiers, and other required information. The first activity agent in the process, say the *receive* activity in the process in Figure 2, receives this publication, instantiates a process instance, processes the activity, and triggers the successor *assign* activity. Agents execute and trigger one another using pub/sub messages in this event-driven manner until the process terminates.

Unlike a centralized orchestration engine, the NIÑOS agent-based engine supports flexible deployment. All activity agents can be deployed at one node, effectively executing processes in a centralized manner, or distributed across the network to realize fully distributed execution. It is also possible to cluster sets of agents and to achieve partially distributed execution. By automatically and dynamically deploying agents at strategic points in the network based on network conditions and available system resources, the NIÑOS execution engine can optimize the business processes. Such QoS-based business process execution is one of the ongoing research directions for this system [Chau et al. 2008; Muthusamy et al. 2009].



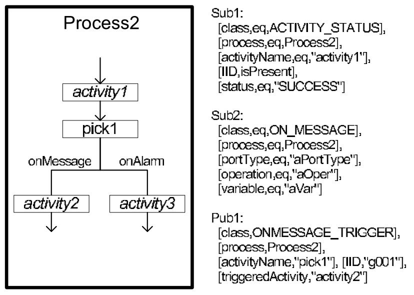
```
Sub1:
  [class,eq,ACTIVITY_STATUS],
  [process,eq,Process1],
  [activityName,eq,"activity1"],
  [IID,isPresent],
  [status,eq,"SUCCESS"]

Sub2:
  [class,eq,ACTIVITY_STATUS],
  [process,eq,Process1],
  [activityName,eq,"activity3"],
  [IID,isPresent],
  [status,eq,"SUCCESS"]

Pub1:
  [class,WHILE_TRIGGER],
  [process,Process1],
  [activityName,"while1"], [IID,"g001"],
  [triggeredActivity,"activity2"]

Pub2:
  [class,ACTIVITY_STATUS],
  [process,Process1],
  [activityName,"while1"], [IID,"g001"],
  [status,"FINISHED"]
```

Fig. 4. BPEL while activity



```
Sub1:
  [class,eq,ACTIVITY_STATUS],
  [process,eq,Process2],
  [activityName,eq,"activity1"],
  [IID,isPresent],
  [status,eq,"SUCCESS"]

Sub2:
  [class,eq,ON_MESSAGE],
  [process,eq,Process2],
  [portType,eq,"aPortType"],
  [operation,eq,"aOper"],
  [variable,eq,"aVar"]

Pub1:
  [class,ONMESSAGE_TRIGGER],
  [process,Process2],
  [activityName,"pick1"], [IID,"g001"],
  [triggeredActivity,"activity2"]
```

Fig. 5. BPEL pick activity

The PADRES and NIÑOS system architecture is conceptually summarized in Figure 3. A set of computing and network resources are virtualized by the PADRES distributed content-based pub/sub routing layer. Over this layer a set of distributed NIÑOS agents collaborate and coordinate to execute a business process. Finally, various tools are available to monitor and manage the process execution and the pub/sub layer.

3.3 Process transformation

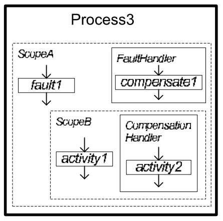
NIÑOS supports the transformation of the complete set of BPEL features, including fault, compensation, and event handling. This section outlines the transformation of some of the more interesting BPEL activities from Table I, notably the *while*, *pick*, *compensate*, *switch*, and *flow* activities.

3.3.1 *While activity.* The BPEL *while* activity repeatedly executes a sequence of activities until a condition, which is a Boolean expression on BPEL variables, is no longer satisfied.

A generic use of the *while* activity is shown in the BPEL process fragment in Figure 4, where the italicized activities are placeholders for one of the standard BPEL activities. The *while* activity is mapped to a *while* agent that evaluates the condition expressed in the activity, and triggers the appropriate subsequent activity. In NIÑOS, the *while* agent evaluates the while condition at the beginning of each iteration of the loop. In order to be triggered at this time, the *while* agent issues the subscriptions Sub1 and Sub2 in Figure 4. These subscriptions are matched by the successful completion of the activity preceding the *while* activity, or by the final activity within the while loop.

As well, the *while* agent issues publications Pub1 and Pub2 in Figure 4 to trigger another iteration of the while loop or to exit the loop and continue execution with the first activity following the loop.

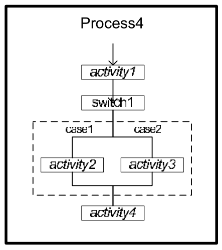
Although not shown, the *while* agent mapping also specifies the subscription and publication messages for the activity preceding the *while* activity (activity1 in Figure 4), the first and last activities within the while loop (activity2 and activity3), and the first activity after the loop (activity4). Also not shown are the messages



```
Sub1:
  [class,eq,COMP_INFO],
  [process,Process3],
  [activityName,eq,"compensate1"],
  [IID,isPresent,"g001"],
  [compScopeName,eq,"ScopeB"]

Pub1:
  [class,COMP_HANDLER_TRIGGER],
  [process,Process3],
  [activityName,"ScopeB"],
  [IID,"g2"],
  [triggeredActivity,"activity2"]
```

Fig. 6. BPEL compensate activity



```
Sub1:
  [class,eq,ACTIVITY_STATUS],
  [process,eq,Process4],
  [activityName,eq,"activity1"],
  [IID,eq,SX],
  [status,eq,"SUCCESS"]
          &&
  [class,eq,VARIABLE_UPDATE],
  [process,eq,Process4],
  [IID,eq,SX],
  [variableName,eq,"var1"]
          &&
  ......     ......
          &&
  [class,eq,VARIABLE_UPDATE],
  [process,eq,Process4],
  [IID,eq,SX],
  [variableName,eq,"varN"]

Pub1:
  [class,SWITCH_CASE_TRIGGER],
  [process,eq,Process4],
  [activityName,"switch1"],
  [IID,"g001"],
  [triggeredActivity,"activity2"]

Sub2:
  [class,eq,SWITCH_CASE_TRIGGER],
  [process,eq,Process4],
  [activityName,eq,"switch1"],
  [IID,isPresent],
```

Fig. 7. BPEL switch activity

used to assign and retrieve variables. For example, the *while* activity may subscribe to update publications for any variables used in the while condition. The handling of BPEL variables is discussed further in Section 3.3.7.

3.3.2 *Pick activity.* The BPEL *pick* activity waits for one or more events to occur and conditionally executes a sequence of activities based on the event that occurred. The events a *pick* activity can wait on include messages, such as Web service invocations or asynchronous replies, and alarms, which are triggered after some time duration or deadline.

A generic use of the *pick* activity is shown in Figure 5. Note that many details, such as the onMessage parameters, are omitted. The *pick* activity is mapped to a *pick* agent that blocks and listens for one of the events specified in the pick activity to occur, and then triggers the appropriate subsequent activity. The execution of the *pick* activity is triggered when the preceding activity complete, which the *pick* agent listens for with subscription Sub1 in Figure 5. Also, the *pick* agent issues a subscription for each onMessage it listens for (Sub2 in Figure 5), and when a matching event occurs, it issues a publication to trigger the appropriate activity (Pub1 in Figure 5).

Note that no subscriptions are issued for onAlarm events since alarm deadlines or durations are evaluated internally by the *pick* agent. As with the previous activity, not all the subscription and publications messages are shown here.

3.3.3 *Compensate activity.* Compensation handlers are an application specific rollback mechanism in BPEL. The activities in a BPEL process are grouped into arbitrarily nested scopes, and each scope may define a *fault handler* and a *compensation handler*. When a fault, or exception, occurs, the scope's fault handler is called. A *compensate* activity within the fault handler can call the compensation handlers for any nested scopes that have successfully executed. A compensation handler attempts to "undo" the logic within the scope. For example, the compensation for a scope whose activities ship a product to a customer may be to cancel the order if it hasn't been delivered yet, or otherwise notify the customer that the order cannot be canceled.

A generic use of the *compensate* activity is shown in Figure 6. Here, ScopeA's fault handler invokes the compensation handler in ScopeB. The *scope* agent for

ScopeB subscribes to compensation events for its scope with Sub1 in Figure 6, and triggers the first activity in its compensation handler using Pub1 in Figure 6.

BPEL semantics require the compensation handler to be called with a snapshot of the variables when the scope completed. This can be achieved by retrieving these values using the PADRES historic access capability [Li et al. 2007], or by having each scope handler cache these values upon scope completion. These cached values would be flushed when the process instance completes. In Figure 6, this would be done by ScopeB's *scope* agent.

3.3.4 *Switch activity.* The BPEL *switch* activity allows for conditional execution, whereby one of several *case* branches is executed based on a Boolean condition associated with each *case*. The cases are ordered and the first branch whose condition evaluates to true is taken. If all the cases fail, an optional *otherwise* branch is taken.

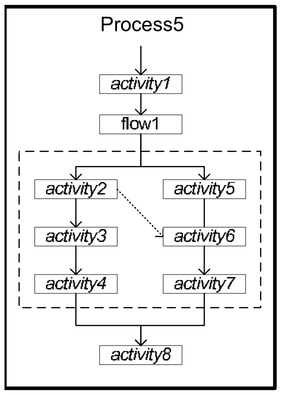
Figure 7 gives an example of a process with a *switch* activity. Not illustrated in the figure is the possibility for execution to transfer directly from the switch1 activity to activity4 if neither case condition is true. In NIÑOS, a *switch* agent is used to evaluate the case conditions in each branch of a *switch* activity.

A *switch* agent subscribes to updates from the system for any variables necessary to evaluate the case conditions, and determines which (if any) branch should be taken. By using a composite subscription (Sub1 in Figure 7), the *switch* agent receives a single notification of its predecessor activity's completion, along with all the required variable updates in the associated process instance. After evaluating the case conditions, the *switch* agent triggers the appropriate branch with a publication such as Pub1 in Figure 7. The first activity in each branch subscribes to these trigger publications. For example, in Figure 7, activity2 subscribes to Sub2. Note that the case where none of the cases in a *switch* activity are taken is not shown.

An alternative implementation could eliminate the need for a *switch* agent entirely, by transferring the responsibility of determining the appropriate branch to follow to the first activities within each case branch. For example, in Figure 7, the agents associated with activity2 and activity3 could independently determine if they should execute. The tradeoff, however, is that these agents will have to perform redundant computations of the case conditions. Recall that the case statements are ordered and only the first true case condition is executed. Therefore, in Figure 7, activity3 must evaluate the condition that case2 is true and that case1 is false. These redundant computations are unnecessary if the conditions are evaluated by a single *switch* agent. Furthermore, distributing the computation of the case conditions requires sending the variable updates necessary to compute these conditions to several agents.

3.3.5 *Flow activity.* The BPEL *flow* activity supports the execution of parallel branches. Branches in a flow typically execute concurrently, but may be synchronized by a *link*. A *link* between a *source* and *target* specifies that the target activity executes only after the source activity has completed. An activity may be the source or target of multiple links.

In addition, a source activity may set a Boolean valued *transition condition* on



```
Sub1:
  [class,eq,ACTIVITY_STATUS],
  [process,eq,Process5],
  [activityName,eq,"activity1"],
  [IID,isPresent],
  [status,eq,"SUCCESS"]

Pub1:
  [class,ACTIVITY_STATUS],
  [process,eq,Process5],
  [activityName,"flow1"],
  [IID,"g001"],
  [status,"STARTED"]

Sub2:
  [class,eq,ACTIVITY_STATUS],
  [process,eq,Process5],
  [activityName,eq,"flow1"],
  [IID,isPresent],
  [status,eq,"STARTED"]

Pub2:
  [class,ACTIVITY_STATUS],
  [process,eq,Process5],
  [activityName,"activity2"],
  [IID,"g001"],
  [status,"SUCCESS"]

Pub3:
  [class,LINK_STATUS],
  [process,eq,Process5],
  [activityName,"activity2"],
  [IID,"g001"],
  [status,"POSITIVE"]

Sub4:
  [class,eq,ACTIVITY_STATUS],
  [process,eq,Process5],
  [activityName,eq,"activity2"],
  [IID,eq,$X],
  [status,eq,"SUCCESS"]
        &&
  [class,eq,LINK_STATUS],
  [process,eq,Process5],
  [activityName,eq,"activity2"],
  [IID,eq,$X],
  [status,isPresent]

Pub4:
  [class,ACTIVITY_STATUS],
  [process,eq,Process5],
  [activityName,"activity6"],
  [IID,"g001"],
  [status,"SUCCESS"]

Sub5:
  [class,eq,ACTIVITY_STATUS],
  [process,eq,Process5],
  [activityName,eq,"activity6"],
  [IID,isPresent],
  [status,eq,"SUCCESS"]
        ||
  [class,eq,ACTIVITY_STATUS],
  [process,eq,Process5],
  [activityName,eq,"activity6"],
  [IID,isPresent],
  [status,eq,"SKIPPED"]

Pub5:
  [class,ACTIVITY],
  [process,eq,Process5],
  [activityName,"activity7"],
  [IID,"g001"],
  [status,"SUCCESS"]
```

Fig. 8. BPEL flow activity

its outgoing links based on an expression of its process instance's state. Likewise, a target activity may specify a Boolean valued *join condition* based on instance state including the state of its incoming links. A target activity executes only if at least one of its incoming links evaluates to true and its join condition is true. A join condition failure, by default, generates a fault, and control is passed to the appropriate fault handler. This fault, however, may be suppressed by setting the *suppressJoinFailure* attribute to true. In the latter case, the target activity is skipped, and all its outgoing links (if any) are set to false.

A generic use of the *flow* activity, including the use of a *link*, is shown in Figure 8. For brevity, not all messages are shown, and notably, transition and join conditions are omitted, and assumed to evaluate to true. The *flow* activity maps to a *flow* agent which waits for the preceding activity to finish (Sub1 in Figure 8), triggers the execution of each flow branch (Pub1 in Figure 8), and then waits for each branch to complete before triggering the subsequent activity.

Activities within a flow are first mapped to NIÑOS agents based on their associated transformation rules. For example, a *flow* activity agent will subscribe to and publish messages as outlined earlier. Then, each activity agent within a flow is augmented with the behavior described in the following paragraphs.

The first activity in each flow branch subscribes to the initiation of the flow (Sub2 in Figure 8), and publishes its completion as usual (Pub2 in Figure 8). Both activity2 and activity5 belong to this case in Figure 8.

Each link source activity publishes the transition condition of each outgoing link. In Figure 8, Pub3 indicates a true transition condition on activity2's outgoing link. On the other hand, link targets subscribe to the status of their incoming links and the source activities associated with those links. For example, in Figure 8, activity6 subscribes to Sub4, and publishes Pub4 when it has completed successfully. A target activity that does not execute, due to a false join condition, publishes that it has skipped the execution of the activity. A successor activity to a link target must, therefore, subscribe to both the execution or suppression of its predecessor. In

Figure 8, activity7, for example, would subscribe to Sub5 and publish Pub5 upon completion. Notice that the use of the *composite subscriptions* feature in Sub4 and Sub5 offloads the detection of event correlation patterns to the PADRES pub/sub layer, simplifying the work of the activity agents.

All other activities publish and subscribe as usual, and do not change their behavior as a consequence of belonging within a *flow*.

Note that the cases above are not mutually exclusive, and an activity may be required to behave according to multiple descriptions. For example, an activity may be both the first activity in a flow branch and the target of a link, or may be both a source and target of (different) links.

3.3.6 *Other activities.* The mappings for the basic BPEL activities from Table I are relatively straightforward. For example, the *reply* activity subscribes to the successful completion of its predecessor activity, and publishes the reply message along with any variable updates. The *fault* activity, likewise, subscribes to the completion of its predecessor activity and publishes a fault message. The mapping of the *sequence* structured activity is also routine compared to the other activities described above. Each activity within a sequence simply subscribes to its predecessor's completion, and publishes its own completion status.

3.3.7 *BPEL variables.* Activities within a BPEL process instance share data by means of *variables*, which are global within a process instance. NIÑOS supports two mechanisms to support BPEL variables.

The first mechanism maintains variables in a distributed manner. Every activity that modifies a variable publishes a *VARIABLE_UPDATE* message with the new value. Any activity that needs to read a variable issues a subscription for these update messages and caches this information locally. In this scheme, each activity agent independently maintains the variable value, and in the case of a sequential process, the value will be consistent across all activities.

A second mechanism addresses the issue of concurrent accesses to variables as is possible with activities executing in parallel flows in a process. In this case, a *variable agent* is used to maintain consistent variable values, and synchronize accesses to variables. Adopting standard distributed locking techniques, activities that read or write to variables must first acquire a read or write lock, respectively, from the variable agent and then retrieve the current variable value from the variable agent. The variable agent supports concurrent reads but exclusive writes. We plan in future work to explore the use of distributed locking algorithms that support greater concurrency and efficiency.

The variable agent mechanism can always be used, while distributed *VARIABLE_UPDATE*s are guaranteed to operate correctly only when variables are not accessed concurrently. Since it is straightforward to distinguish the potentially concurrent and sequential portions of a BPEL process, the process transformation is able to use the distributed *VARIABLE_UPDATE* mechanism in sequential parts of the process, but revert to variable agents in concurrent portions.

The *visibility* of variables by activities in different scopes is well-defined in the BPEL specification, and can be determined and resolved during process transformation. For example, activities would only issue subscriptions for updates to variables

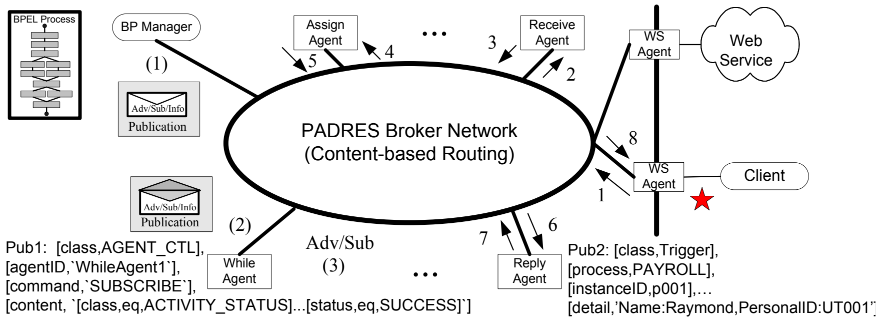

BPEL Process
BP Manager
(1)
Adv/Sub/Info
Publication
Assign Agent
5
4
...
3
Receive Agent
2
WS Agent
Web Service
PADRES Broker Network
(Content-based Routing)
8
WS Agent
Client
1
(2)
Adv/Sub
(3)
While Agent
7
6
Reply Agent
...
Pub1: [class,AGENT_CTL], [agentID,`WhileAgent1`], [command,`SUBSCRIBE`], [content, `[class,eq,ACTIVITY_STATUS]...[status,eq,SUCCESS]`]
Pub2: [class,Trigger], [process,PAYROLL], [instanceID,p001],... [detail,'Name:Raymond,PersonalID:UT001'

Fig. 9. Process deployment and execution monitor

declared within their own or ancestor scopes. Other activities, for whom these variables are not supposed to be visible would not subscribe to and hence would not receive these variable updates.

3.4 Process deployment

The result of process transformation is a set of subscription, advertisement and activity information messages representing the BPEL activities in a business process. The goal of process deployment is to install an activity at a particular agent by sending the advertisements, subscriptions and the activity information generated from the transformation phase to available activity agents in the system.

Exploiting the publish/subscribe paradigm, the process manager wraps the above messages inside envelopes compliant with the publish/subscribe language and sends them to activity agents. The envelopes are agent control publications with class *AGENT_CTL*, and contain the information that the manager wants to deliver to an agent, and the identifier of the particular agent in the *agentID* predicate. Activity agents receive the control messages by subscribing to *AGENT_CTL* messages addressed to themselves. Upon receiving an envelope, an agent unwraps the enclosed message and issues the messages as its own subscriptions or advertisements, as shown in Figure 9.

The process of installing an activity at an agent consists of five steps. First, a set of subscription, advertisement and activity information messages are generated from a business process definition file during the process transformation phase. Second, the messages are wrapped in an envelope as a field of an *AGENT_CTL* publication. Third, the publish/subscribe broker network delivers the *AGENT_CTL* publications to the addressed agents. Fourth, the agent extracts the subscription, advertisement, and activity information messages from the *AGENT_CTL* message. For instance, Pub1 in Figure 9 is an *agent control* publication wrapping a subscription for the *while* agent. Finally, the agent processes the messages based on the *command* field which has three possible values: *subscribe*, *advertise* or *activityinfo*. The *subscribe* command causes the agent to subscribe to the subscription specified in the *content* field, and not surprisingly, the *advertise* command causes the agent to advertise the advertisement contained in the *content* field. Whereas subscriptions and advertisements describe the activity dependency of a process, the *activityinfo* control message contains information needed by an activity agent to execute the

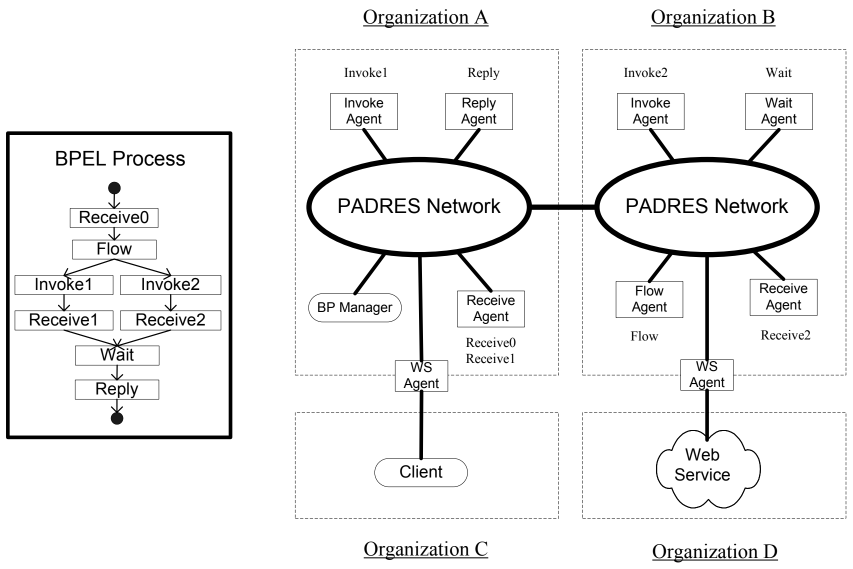


Fig. 10. Inter-organizational deployment

activity, such as the Boolean looping condition for a *while* activity. At this point, the business process is deployed and each agent is ready for execution.

We emphasize that after a BPEL process has been transformed into advertisements, subscriptions and activityinfo messages, there is much flexibility in the activity agents where these messages are installed. Furthermore, the provisioning of the quantity and types of activity agents can itself be arbitrary and accommodate to system requirements. For example, Figure 10 shows a scenario where organizations A and B decide to collaborate in hosting a business process. Each organization administers its own PADRES federation, and decides on which set of activity agents to provision. Notice that there may be multiple agents of the same type. Such replication of activity agents allows greater flexibility during process deployment, provides more resources with which to balance and support greater loads, and supports redundancy in case of failures. The BPEL process in Figure 10 may be deployed to the activity agents as annotated in the figure. Notice that regardless of the complexity of the network architecture, activity agents are simply identified by their location-independent address in the PADRES network, and the deployment of a BPEL process to activity agents proceeds exactly as above. As elaborated in Section 3.6, the ability of the PADRES content-based pub/sub layer to address components in the system by their network- and location-independent name is key to managing the complexity of arbitrarily elaborate deployments.

While organizations that wish to participate in the execution of a BPEL process must administer a PADRES/NIÑOS deployment, it remains possible to invoke processes hosted by other organizations (see Organization D in Figure 10) that expose their processes as Web services. Furthermore, BPEL processes executed by the distributed NIÑOS system can be invoked by outside clients (see Organization C in Figure 10). The scenario in Figure 10 illustrates the flexible deployment options available to organizations in terms of the distribution of the NIÑOS execution engine, and interactions with business partners and clients. The determination of an appropriate or optimal deployment is driven by business policies and goals and is the subject of ongoing work.

3.5 Process execution

The activity agents attached to the publish/subscribe system are responsible for executing the activities in the process. They are both subscribers and publishers, subscribing to activity completion events from predecessor activities and publishing events to notify their successor activities. The dual roles enable them to exchange messages within the publish/subscribe messaging system, allowing coordinated execution of the business process.

A particular instance of a process is started by a NIÑOS service request, such as Pub2 in Figure 9, and is driven by activity completion events. Execution continues until all the activities defined in the process are finished. The process flow, or dependencies between activities, is encoded in the interplay between subscriptions and advertisements, which determine the order of activity execution. Dependency subscriptions may be composite subscriptions, in which case matching is performed in the broker network, and agents are notified only when their execution conditions are fully satisfied. Detecting the execution condition in the PADRES broker network makes the activity agent a light-weight component in NIÑOS without significant processing or storage requirements. During execution, all the message routing is automatic and transparent to the process management layer.

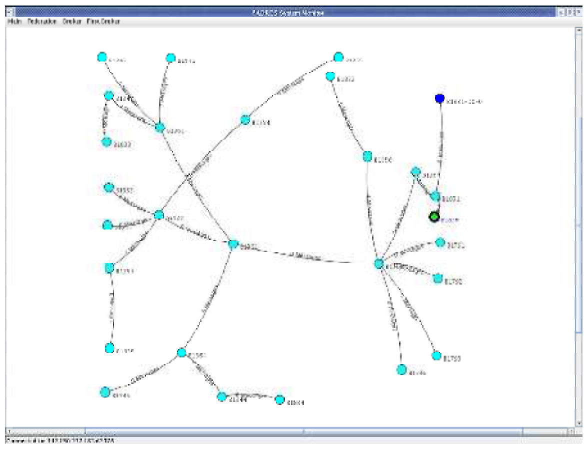
3.6 Process management

Enterprises demand powerful facilities to control and monitor their business processes. Convenient management features are even more important in distributed architectures. We highlight a few management scenarios below, and describe how they are supported in NIÑOS.

NIÑOS provides a graphical monitoring interface to visualize the network topology, message routing, and distributed process execution, as shown in Figure 11. The monitoring itself is entirely based on pub/sub messages, making it possible, for example, to observe what others are monitoring.

Both real-time and historic process monitoring are supported by NIÑOS. Real-time monitoring is simple to achieve in NIÑOS due to the use of a content-based pub/sub infrastructure. The monitor, shown in Figure 11, which is itself a pub/sub client, subscribes to the execution information of a particular activity, allowing the monitor to know the execution status of a process. The expressive content-based pub/sub semantics allow the monitor to observe the status of individual activities, trace the execution of a particular process instance, or perform countless other queries, all without requiring additional instrumentation logic at the components being monitored. For example, the first $ACTIVITY\_STATUS$ subscription in Figure 11 allows an administer to view, in real-time, the operation of a particular while activity agent in the system including the invocations of the activity.

Enterprise applications also require probing the execution of completed processes, perhaps for auditing or analysis purposes. The PADRES infrastructure supports historic data access using subscriptions that unify the query for past and future



Monitor an activity:
[class,eq,ACTIVITY_STATUS],
[process,eq,Process1],
[activityName,eq,"while"], [IID,isPresent],
[status,isPresent]

Trace a faulty process instance:
[class,eq,ACTIVITY_STATUS],
[activityName,eq,"activity2"], [IID,eq,$X]
[status,eq,"FAILURE"]
&&
[class,eq,ACTIVITY_STATUS],
[activityName,isPresent], [IID,eq,$X]

Pause a process instance:
[class,AGENT_CTL],
[command,PAUSE], [agentID,isPresent],
[process,"PAYROLL"], [IID,p001]

Fig. 11. PADRES monitor

events [Li et al. 2007]. Along with PADRES's composite subscriptions feature [Li and Jacobsen 2005], both executing and previously executed process instances can be correlated and queried. For example, it is possible to monitor the status of new process invocations by users who invoked the process at least ten times yesterday.

Another management scenario is to trace the execution of process instances that exhibit some behavior. For example, the second set of *ACTIVITY_STATUS* subscriptions[1] trace the invocations of every activity for those process instances whose *activity2* failed. Examining the execution of these instance can help diagnose the failure or understand its consequences.

Advanced process control functions include suspending, resuming or stopping running process instances. The target instances can be specified by instance id, process id, or any constraints expressible by the pub/sub semantics. For example, the *AGENT_CTL* publication in Figure 11 instructs all agents executing the *PAYROLL* process to suspend the execution of instance *p001*.

These functions are useful especially when processes need to be updated online. For example, a manager may suspend running process instances, dynamically update certain activities in the process (by sending modified subscription, advertisement, and activityinfo envelopes to activity agents), and resume the instances. The agent-based execution in NIÑOS simplifies this task since only the agents corresponding to the modified activities need to participate in the process redefinition. The other activities can continue executing the process.

As mentioned earlier, the provisioning of multiple instances of the same activity agent type provides more process deployment choices, greater scalability potential, and the ability to redeploy the activities assigned to a failed activity agent. For example, in Figure 10, if the receive agent provisioned by Organization A fails, the *Receive0* and *Receive1* activities can be redeployed to the receive agent provisioned by Organization B using the management features described above. While the mechanisms required to respond to failures are supported by NIÑOS, the automatic

[1]We use a composite subscription in this case.

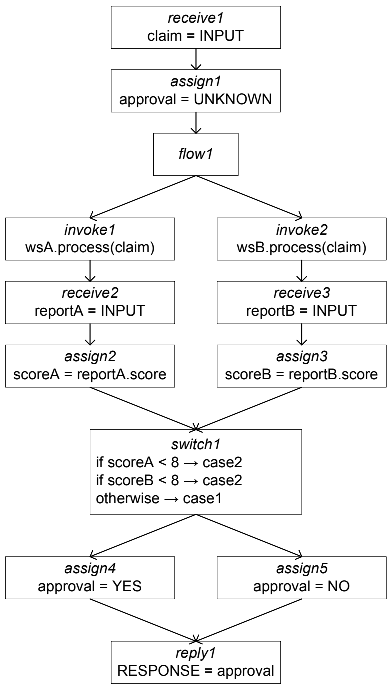


Fig. 12. Example loan application process

detection and correction of failures is left for future work. Towards this end, we have investigated failure resilience in the PADRES network layer [Kazemzadeh and Jacobsen 2009], and formalized well-defined semantics for the mobility of activity agents [Hu et al. 2009].

3.7 Example

Consider a loan approval BPEL process in Figure 12. The process is triggered when a loan application is received. In order to avoid approving risky loans, the process invokes two external Web services that independently generate a credit report for the loan applicant. Only if both credit rating services deem the applicant to be credit worthy does the process approve the loan application.

Each activity in the BPEL process in Figure 12 is mapped to a NIÑOS agent, and table II details the advertisements and subscriptions issued by each agent, as well the publications for a sample run of the process. Some of the agents in the parallel branches of the process, such as the invoke2 and receive3 activities, are omitted from Table II. Their messages would correspond to the messages issued by the corresponding activities in the other branch.

Although there is a flow activity in the BPEL process in Figure 12, there is no corresponding agent. Instead, the first activity in each branch of the flow are triggered when the final activity before the flow activity completes. Similarly, it is possible to eliminate the switch activity by having the first activities in each branch of the switch subscribe to their respective conditions directly. The switch activity is assigned to an agent in Table II to illustrate what its message would look like.

Activity	**Subscription**	**Advertisement**	**Sample Publication**
(trigger)	-	[class,eq,TRIGGER], [process,eq,"loan"], [IID,isPresent]	[class,TRIGGER], [process,"loan"], [IID,"g001"], <<CLAIM>>
receive1	[class,eq,TRIGGER], [process,eq,"loan"], [IID,isPresent]	[class,eq,ACTIVITY_STATUS], [process,eq,"loan"], [activityName,eq,"receive1"], [IID,isPresent], [status,isPresent] [class,eq,VARIABLE_UPDATE], [process,eq,"loan"], [activityName,eq,"receive1"], [IID,isPresent], [variableName,eq,"claim"]	[class,ACTIVITY_STATUS], [process,"loan"], [activityName,"receive1"], [IID,"g001"], [status,"SUCCESS"] [class,VARIABLE_UPDATE], [process,"loan"], [activityName,"receive1"], [IID,"g001"], [variableName,"claim"], <<CLAIM>>
assign1	[class,eq,ACTIVITY_STATUS], [process,eq,"loan"], [activityName,eq,"receive1"], [IID,isPresent], [status,eq,"SUCCESS"]	[class,eq,ACTIVITY_STATUS], [process,eq,"loan"], [activityName,eq,"assign1"], [IID,isPresent], [status,isPresent] [class,eq,VARIABLE_UPDATE], [process,eq,"loan"], [activityName,eq,"assign1"], [IID,isPresent], [variableName,eq,"approval"]	[class,ACTIVITY_STATUS], [process,"loan"], [activityName,"assign1"], [IID,"g001"], [status,"SUCCESS"] [class,VARIABLE_UPDATE], [process,"loan"], [activityName,"assign1"], [IID,"g001"], [variableName,"approval"], <<RESULT>>
flow1	-	-	-
invoke1	[class,eq,ACTIVITY_STATUS], [process,eq,"loan"], [activityName,eq,"assign1"], [IID,eq,$X], [status,eq,"SUCCESS"] && [class,eq,VARIABLE_UPDATE], [process,eq,"loan"], [activityName,eq,"receive1"], [IID,eq,$X], [variableName,eq,"claim"]	[class,eq,ACTIVITY_STATUS], [process,eq,"loan"], [activityName,eq,"invoke1"], [IID,isPresent], [status,isPresent]	[class,ACTIVITY_STATUS], [process,"loan"], [activityName,"invoke1"], [IID,isPresent], [status,"SUCCESS"]
wsA	[class,eq,ACTIVITY_STATUS], [process,eq,"loan"], [activityName,eq,"invoke1"], [IID,eq,$X], [status,eq,"SUCCESS"] && [class,eq,VARIABLE_UPDATE], [process,eq,"loan"], [activityName,eq,"receive1"], [IID,eq,$X], [variableName,eq,"claim"]	[class,eq,ACTIVITY_STATUS], [process,eq,"loan"], [activityName,eq,"wsA"], [IID,isPresent], [status,isPresent]	[class,ACTIVITY_STATUS], [process,"loan"], [activityName,"wsA"], [IID,eq,"g001"], [status,eq,"SUCCESS"], <<RESULT>>
receive2	[class,eq,ACTIVITY_STATUS], [process,eq,"loan"], [activityName,eq,"wsA"], [IID,isPresent], [status,eq,"SUCCESS"]	[class,eq,ACTIVITY_STATUS], [process,eq,"loan"], [activityName,eq,"receive2"], [IID,isPresent], [status,isPresent] [class,eq,VARIABLE_UPDATE], [process,eq,"loan"], [activityName,eq,"receive2"], [IID,isPresent], [variableName,eq,"reportA"]	[class,ACTIVITY_STATUS], [process,"loan"], [activityName,"receive2"], [IID,"g001"], [status,"SUCCESS"] [class,VARIABLE_UPDATE], [process,"loan"], [activityName,"receive2"], [IID,"g001"], [variableName,"reportA"], <<RESULT>>

assign2	[class,eq,ACTIVITY_STATUS], [process,eq,"loan"], [activityName,eq,"receive2"], [IID,eq,$X], [status,eq,"SUCCESS"] && [class,eq,VARIABLE_UPDATE], [process,eq,"loan"], [activityName,eq,"receive2"], [IID,eq,$X], [variableName,eq,"reportA"]	[class,eq,ACTIVITY_STATUS], [process,eq,"loan"], [activityName,eq,"assign2"], [IID,isPresent], [status,isPresent] [class,eq,VARIABLE_UPDATE], [process,eq,"loan"], [activityName,eq,"assign2"], [IID,isPresent], [variableName,eq,"scoreA"]	[class,ACTIVITY_STATUS], [process,"loan"], [activityName,"assign2"], [IID,"g001"], [status,"SUCCESS"] [class,VARIABLE_UPDATE], [process,"loan"], [activityName,"assign2"], [IID,"g001"], [variableName,"scoreA"], <<APPROVAL>>
switch1	[class,eq,ACTIVITY_STATUS], [process,eq,"loan"], [activityName,eq,"assign2"], [IID,eq,$X], [status,eq,"SUCCESS"] && [class,eq,VARIABLE_UPDATE], [process,eq,"loan"], [activityName,eq,"assign2"], [IID,eq,$X], [variableName,eq,"scoreA"] && [class,eq,ACTIVITY_STATUS], [process,eq,"loan"], [activityName,eq,"assign3"], [IID,eq,$X], [status,eq,"SUCCESS"] && [class,eq,VARIABLE_UPDATE], [process,eq,"loan"], [activityName,eq,"assign3"], [IID,eq,$X], [variableName,eq,"scoreB"]	[class,eq,SWITCH_CASE_TRIGGER], [process,eq,"loan"], [activityName,eq,"switch1"], [IID,isPresent], [triggeredActivity,isPresent]	[class,SWITCH_CASE_TRIGGER], [process,"loan"], [activityName,"switch1"], [IID,"g001"], [triggeredActivity,"assign4"]
assign4	[class,eq,SWITCH_CASE_TRIGGER], [process,eq,"loan"], [activityName,eq,"switch1"], [IID,isPresent], [triggeredActivity,eq,"assign4"]	[class,eq,ACTIVITY_STATUS], [process,eq,"loan"], [activityName,eq,"assign4"], [IID,isPresent], [status,isPresent] [class,eq,VARIABLE_UPDATE], [process,eq,"loan"], [activityName,eq,"assign4"], [IID,isPresent], [variableName,eq,"approval"]	[class,ACTIVITY_STATUS], [process,"loan"], [activityName,"assign4"], [IID,"g001"], [status,"SUCCESS"] [class,VARIABLE_UPDATE], [process,"loan"], [activityName,"assign4"], [IID,"g001"], [variableName,"approval"], <<YES>>

reply1	([class,eq,ACTIVITY_STATUS], [process,eq,"loan"], [activityName,eq,"assign4"], [IID,eq,$X], [status,eq,"SUCCESS"] && [class,eq,VARIABLE_UPDATE], [process,eq,"loan"], [activityName,eq,"assign4"], [IID,eq,$X], [variableName,eq,"approval"]) || ([class,eq,ACTIVITY_STATUS], [process,eq,"loan"], [activityName,eq,"assign5"], [IID,eq,$X], [status,eq,"SUCCESS"] && [class,eq,VARIABLE_UPDATE], [process,eq,"loan"], [activityName,eq,"assign5"], [IID,eq,$X], [variableName,eq,"approval"])	[class,eq,REPLY], [process,eq,"loan"], [IID,isPresent], [status,isPresent]	[class,REPLY], [process,"loan"], [IID,"g001"], [status,"SUCCESS"], <<approval>>

Table II: Agent messages for loan application process in Figure 12

3.8 Architectural benefits

There are several capabilities offered by the distributed NIÑOS execution engine that are either not present in traditional centralized processing architectures or are more difficult to achieve. This section points out some of these qualitative benefits, deferring the quantitative performance benefits to Section 4.

One benefit is that fine-grained monitoring of a running process requires no additional effort, and little overhead. Since process activities are triggered by ordinary pub/sub messages, it is possible to non-intrusively subscribe to these messages and make detailed observations of running processes such as an activity's invocation rate, a branch's execution probabilities, or a process's critical path. The monitoring does not require adding any additional instrumentation code to the process, and the multicast message propagation in PADRES ensures that only the minimal number of additional messages are sent over the network.

The NIÑOS architecture also naturally supports cross-organizational processes, where portions of a process span different administrative domains. For example, the portion of a loan application process that accesses confidential customer credit information may be required to execute within the accounting department. In NIÑOS, the relevant activities can easily be deployed on the resources administered by the appropriate department.

Related to the above point, NIÑOS also supports both orchestration and choreography styles of process execution. In fact, in NIÑOS a BPEL orchestration is mapped into a choreography involving a number of agents.

The NIÑOS architecture supports the capability to modify portions of a process while it is still executing. For example, the processing logic of a particular activity can be changed by deploying a replacement activity agent, having the new agent issue the necessary subscriptions and advertisements, and have the original agent unsubscribe. It is even possible to modify the control logic of a portion of a process using the same techniques to insert a new process fragment into an existing process. Since the process is distributed, this process modification technique allows the remainder of the process to continue executing while one portion of it is being altered.

The NIÑOS execution engine exploits the PADRES complex event processing capabilities to offload certain process execution tasks to the PADRES broker network. For example, activities that are triggered by multiple publications issue a composite subscription for these publications. The publications that contribute to matching the composite subscription are collected and correlated in the broker network itself. This benefits the agents who can avoid processing cost of the correlation, and reduces network traffic by letting the broker network decide the optimal point to collect and correlate the publications.

The decomposition of a process into fine-grained components affords more precise control over load balancing or replication needs. For example, a single activity in a process may be the bottleneck that limits the processing time of the process. Instead of replicating the entire process, only the bottleneck activity needs to be duplicated. The details of realizing this are out of the scope of this article, but are made possible by the distributed NIÑOS execution architecture.

The distributed execution of activities in a process is also potentially more scalable by taking advantage of available distributed resources. Furthermore, due to the fine granularity of the individual execution agents, the system is able to utilize even relatively small resources. For example, certain activities in a process may be very lightweight and the associated agent could be deployed on a relatively underpowered machine; it is not necessary to find a machine that can execute the entire process.

One potential critique of the NIÑOS architecture is that it requires each organization to deploy a federation of PADRES brokers. However, this is conceptually no different from a process choreography where multiple organizations collaborate to execute a business process. In such choreography scenarios, the process spans administrative domains and there is no centralized coordinator, perhaps because the organizations cannot agree on one trusted central entity. Instead, each organization administers its own process execution engine, with standards such as BPMN [White 2004] and the family of Web service specifications facilitating the interoperability among the participants. In a similar manner, the brokers in the NIÑOS architecture can use messaging and event processing standards such as the Java Messaging Service (JMS), Advanced Message Queuing Protocol (AMQP), or WS-Notification allowing each organization to deploy their choice of technology. It should also be reiterated that it is perfectly sensible to deploy the NIÑOS architecture on a single machine and only add additional resources as required.

Many of the benefits of the NIÑOS architecture stem from the distributed nature of the execution engine, where a large process is broken down into fine-grained activities which are each executed by an autonomous agent.

4. EVALUATION

This section quantitatively evaluates the distributed NIÑOS process execution architecture. In particular, it is compared to centralized and clustered architectures. A variety of parameters are varied to attempt to understand the conditions for which each architecture is well suited.

4.1 Experimental setup

NIÑOS is implemented in Java over the PADRES distributed content-based pub/sub system. The evaluations are conducted in a dedicated network of 20 machines, each equipped with 4GB of memory and 1.86GHz CPUs. In all the tests, in addition to the deployed activity agents, there is a process management client, and a service request client that invokes process instances. Since there are no accepted benchmarks in this field, the delivery service business process described in Figure 2 is used.

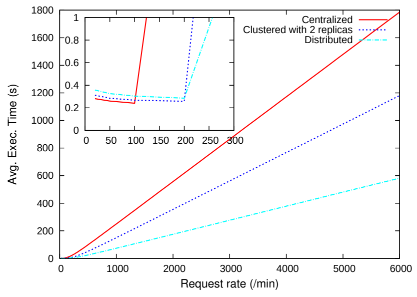
The centralized, clustered and distributed execution deployments are compared. In the centralized scenario, activity agents reside on the same machine, connecting to a single PADRES broker. This deployment serves as a baseline and emulates a centralized execution engine. The clustered scenario attempts to increase the scalability of the system by deploying multiple centralized engines. In the evaluations, two sets of activity agents are deployed, with each set of agents connected to a different broker. A load balancer directs requests evenly across the two process engines, each of which independently execute the requests. For the distributed scenario, a 30 broker network is deployed with the agents connecting to the various brokers. The broker network has a maximum diameter of six hops, and there are 12 edge brokers and 18 inner brokers, the latter of which have a degree between two and four.

In all three deployments, two Web service agents act as proxies to the two external Web services invoked by the process. Since the availability of external Web services is independent of the architecture of the execution, in all three deployments, the number of Web services is fixed at two. Notably, even though multiple copies of activity agents are deployed for the clustered scenario, these agents still share the two external Web services.

The metrics of interest are the average process execution time and the average system throughput while varying the request rate, the delay of the external Web services, and the size of the messages. The process execution time is defined as the duration from the issue of a request by a client to the receipt of the corresponding response by the client, and the throughput is the number of process instances completed per minute. The default values for request rate, Web service execution time, and message size are 500/min, 100 *ms*, and 512 *bytes*, respectively.

4.2 Request rate

This experiment varies the process invocation rate, where each invocation generates a process instance, and measure the average execution time and throughput. As shown in Figure 13, for lower request rates, the centralized approach offers the best execution time, with improvements of 9% and 20% over the clustered and the distributed deployments, respectively. This is because the overhead of the



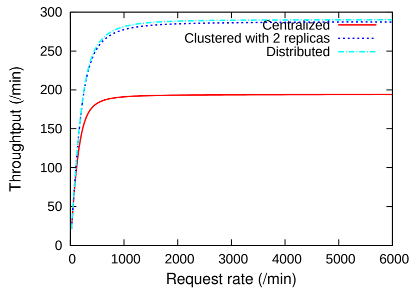
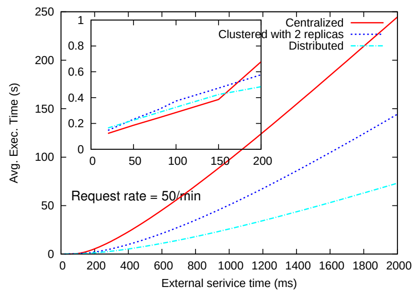


Fig. 13. Varying request rates



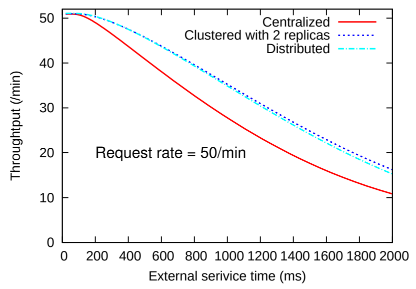


Fig. 14. Varying Web service execution time (50 requests/min)

workload balancer in the clustered approach and the communication overhead of traversing the broker network in the distributed setup are not negligible. When the request rates are higher than 300/min, however, the clustered and the distributed approaches are faster, with 34% and 67% better execution times, respectively, over the centralized scenario at the highest request rate of 6000/min.

The throughput results in Figure 13 show that the distributed and clustered approaches, whose maximum throughput are similar, outperform the centralized one, with a roughly 49% increase in maximum throughput. Notice that for low request rates of around 200/min, the throughput of all the three approaches are almost the same as the request rate because none of the approaches reach their maximum throughput. The virtually identical throughput at higher request rates for the clustered and distributed deployments is because the external Web services invoked by the process are the bottleneck. Since there are no replicas of these Web services, and they are shared among the process instances, the Web services limit the maximum throughput.

Note that a Web service behaving as a throughput bottleneck does not imply it is also a latency bottleneck. For example, even if the processing rate of a Web service is bound, the additional time it takes to execute the remainder of the activities in the process will differ in the distributed and clustered architectures. Therefore, the latency and throughput results in Figure 13 are not inconsistent.

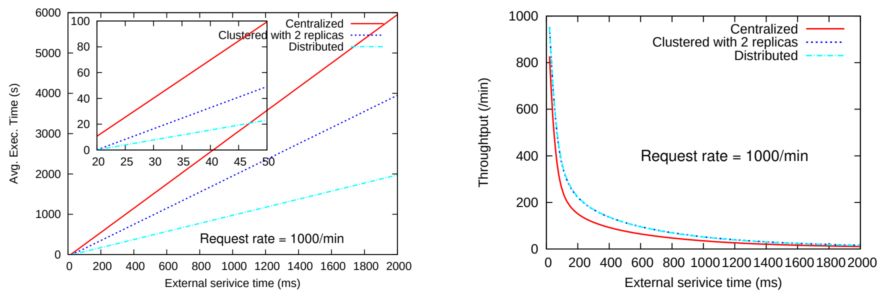


Fig. 15. Varying Web service execution time (1000 requests/min)

4.3 Web service delay

To better understand the effects of the external Web services on both process execution time and throughput, this experiment varies the Web service delay from 20 *ms* to 2 *sec* with two different request rates. With a lower request rate of 50/min, the results in Figure 14 show that, as expected, a longer Web service delay increases the average execution time for all three deployment scenarios. When the delay is small, the centralized approach performs the best by avoiding the communication overhead present in the other two approaches. On the other hand, when the Web service delay increases, the distributed approach performs the best, with 49% and 70% improvements in execution time over the clustered and the centralized scenarios, respectively.

A large Web service delay requires the execution engine to handle many concurrent process instances, which increases the memory and processing requirements on the system. The increased number of process instances are balanced among two clusters in the clustered scenario, resulting in up to a 41% improvement in execution time compared to the centralized approach.

With a much higher request rate of 1000/min, the results in Figure 15 show that the centralized approach performs the worst and the distributed deployment the best regardless of the Web service delay. Recall from Figure 13 that all three deployment scenarios achieved their maximum throughput at a request rate of approximately 1000/min. At this point process instances begin to queue up at activity agents and Web services, and the contribution of this queueing delay on the process execution time dominates the communication overhead present in the clustered and distributed deployments. As in the case with a lower request rate, the clustered and distributed deployments disperse the request load among the additional resources available to them and achieve faster execution times than the centralized scenario. As well, the throughput results in Figure 15 are consistent with the observations from Figure 13, with the throughput benefits of the distributed and clustered deployments disappearing when the Web service delays are increased to a point where the Web service becomes a bottleneck.

4.4 Message size

The amount of data transferred between activities in a BPEL process may vary. This experiment investigates the impact of the inter-activity message size for the

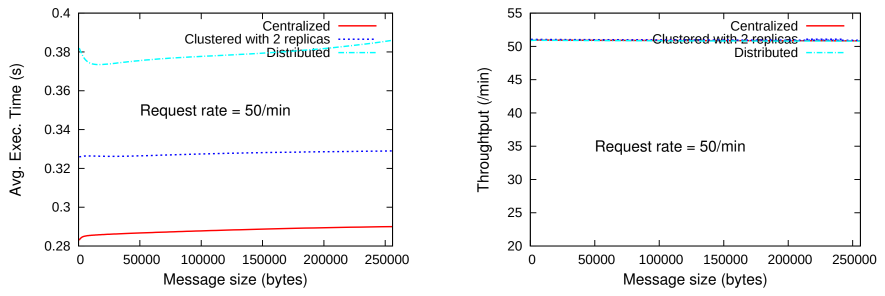


Fig. 16. Varying message size (50 requests/min)

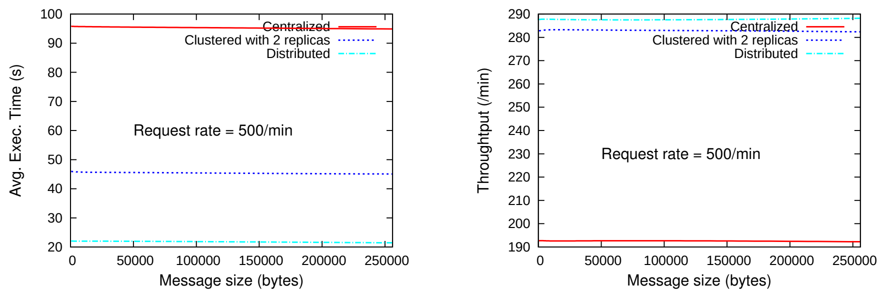


Fig. 17. Varying message size (500 requests/min)

three deployment scenarios. The results in Figures 16 and 17 indicate that varying the message size, even with different request rates, has little effect on either the process execution time or the throughput in any of the three deployment scenarios. However, the distributed case performs the worst with a request rate of 50/min because of the communication overhead and performs the best with a request rate of 500/min because the request queueing times dominate the communication overhead.

When the request rate is low (Figure 16), all three deployment scenarios maintain a throughput that roughly equals the request rate. However, the centralized deployment becomes overloaded with higher requests rates (Figure 17), with the clustered and distributed approaches achieving about 48% better throughput figures. As with latency, the results show that the communication and processing overheads of traversing a larger broker network is not significant with message sizes up to 256 *kbytes*.

Larger messages influence performance in two key ways: an increase in the network overhead when transmitting messages, and an increase in the computation overhead when processing messages. Now, it is not clear to what extent the stable results in Figures 16 and 17 are generalizable to WAN deployments with slower network links, but the compute resources of the machines in the experiment are not unreasonable in commodity hardware, let alone enterprise servers. Therefore, the observation that the processing overhead is largely independent of the message sizes evaluated is likely a more universal phenomenon. This is understood by noticing that a significant portion of the processing overhead is attributable to the pub/sub

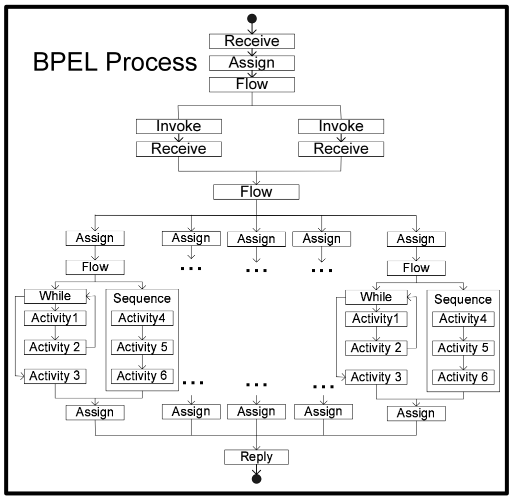


Fig. 18. Parallel business process

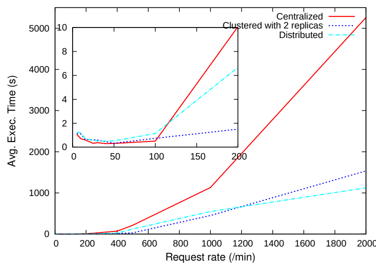
matching of the messages, and the PADRES matching engine we use only performs matching on the attributes and predicates in the message header; the payload is not processed. To exploit this, the BPEL process transformation in NIÑOS only encodes the process control flow details as pub/sub attributes and predicates, and leaves the process data in the payload. In particular, *VARIABLE_UPDATE* publications include the variable's name as a pub/sub attribute, but store the variable's value in the message payload. In this way, variations in message size caused by variable data values do not significantly influence the pub/sub matching time.

4.5 Parallelism

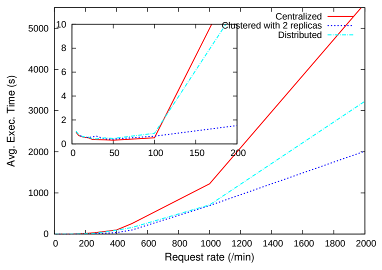
As processes may exhibit different degrees of parallelism, this experiment compares two processes: one containing many activities with ten parallel branches, as shown in Figure 18, and another with the same number activities but with only two parallel branches. To isolate the effects of process parallelism, no external Web services are invoked by either process.

With the highly parallel process, the distributed deployment offers significantly better execution time performance as shown in Figure 19(a). When the request rate is less than 100/min, we observe a trend similar to Figure 13, where the distributed approach performs worse because of the additional network overhead. This is understandable since higher request rates result in more activities executing in parallel, and more opportunities for the distributed deployment to take advantage of the additional resources available to it. At the highest request rates evaluated, the distributed scenario executed the parallel process 79% faster than the centralized approach, and the clustered deployment improved over the centralized one by about 71%.

With the more sequential process, on the other hand, the benefits of additional resources diminishes. It turns out that there is not enough parallelism in this particular process for the distributed deployment to benefit from, and the clustered



(a) Parallel



(b) Sequence

Fig. 19. Average execution time

approach actually achieves the best execution time. The results in Figure 19(b) indicate that at the request rate of 2000/min, the clustered approach is 64% faster than the centralized one, whereas the distributed deployment is only 42% better.

Since there are no Web service requests in the processes, which might limit the maximum throughput, the clustered approach has the best throughput with 25% and 41% improvement for the highly parallel and the less parallel processes, compared to the distributed approach.

According to the results, the distributed approach is better able to achieve low process execution times with processes characterized by many parallel flows and in situations where the request rates are high. Otherwise, there is not enough parallelism to exploit the distributed resources available and the distribution overhead may actually impair the performance. In such situations the centralized or clustered architectures may perform better.

5. CONCLUSIONS

In this paper, we first we propose a distributed business process execution architecture, based on a pub/sub infrastructure, using light-weight activity agents to carry out business process execution in a distributed environment. The pub/sub layer simplifies the interaction among agents, and reduces the cost of maintaining execution state for running process instances. Second, we describe how BPEL activities can be mapped to pub/sub semantics that realize the process control flow among activity agents. These agents are loosely coupled in the pub/sub layer, which makes our agent-based BPEL engine more flexible and customizable. Third, we present how to deploy processes into the agent network, initiate a process instance, and manage the process execution. The process deployment, execution and management are performed through the pub/sub layer taking advantage of the even-driven and the loosely coupled nature of the pub/sub infrastructure. Finally, we carry out a set of experiments comparing our distributed agent-based engine with a centralized orchestration scenario and a clustered scenario. The evaluation indicates that the benefit of the distributed approach is more apparent under a higher process request workload, say over 300 requests per minute. In addition, the distributed approach is well suited to execute highly parallel processes that are not feasible in a centralized deployment.

For future work, we would like to explore more experiments with larger business processes and broker topologies, and study the movement of activity agents in order to satisfy certain goals or constraints. For example, the average execution time may be minimized by moving tightly coupled activity agents close to one another. Moreover, it is interesting to study more advanced techniques for the validation of BPEL process specifications, such as model checking and simulations for process execution and debugging in the distributed environment.

Acknowledgments

This research was supported by IBM's Center for Advanced Studies and Bell Canada's Bell University Laboratories R&D program, and builds on the PADRES research project sponsored by CA, Sun Microsystems, the Ontario Centers of Excellence, the Canada Foundation for Innovation, the Ontario Innovation Trust, and the Natural Sciences and Engineering Research Council of Canada. We would also like to thank Serge Mankovskii from CA and our colleagues including Balasubramaneyam Maniymaran, Pengcheng Wan, and Chunyang Ye for providing valuable feedback on earlier versions of this manuscript.

REFERENCES

Abadi, D. J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.-H., Lindner, W., Maskey, A. S., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., and Zdonik, S. 2005. The Design of the Borealis Stream Processing Engine. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR 2005)*. Asilomar, CA.

Alonso, G., Agrawal, D., Abbadi, A. E., Mohan, C., Gunthor, R., and Kamath, M. 1995. Exotica/FMQM: A persistent message-based architecture for distributed workflow management. In *Proceedings of the IFIP Working Conference on Information Systems Development for Decentralized Organizations (IFIP 1995)*. Trondheim, Norway.

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

Casati, F. and Discenza, A. 2001. Modeling and managing interactions among business processes. *Journal of Systems Integration 10,* 2 (Apr.), 145–168. Special Issue: Coordination as a Paradigm for Systems Integration.

Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M. J., Hellerstein, J. M., Hong, W., Krishnamurthy, S., Madden, S. R., Reiss, F., and Shah, M. A. 2003. TelegraphCQ: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD 2003)*. San Diego, California.

Chau, T., Muthusamy, V., Jacobsen, H.-A., Litani, E., Chan, A., and Coulthard, P. 2008. Automating SLA modeling. In *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research (CASCON 2008)*. Toronto, Canada.

Fabret, F., Jacobsen, H. A., Llirbat, F., Pereira, J., Ross, K. A., and Shasha, D. 2001. Filtering algorithms and implementation for very fast publish/subscribe systems. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data (SIGMOD 2001)*. Santa Barbara, California, United States.

Fidler, E., Jacobsen, H.-A., Li, G., and Mankovski, S. 2005. The PADRES distributed publish/subscribe system. In *Feature Interactions in Telecommunications and Software Systems VIII (ICFI 2005)*. Leicester, UK.

Hu, S., Muthusamy, V., Li, G., and Jacobsen, H.-A. 2009. Transactional mobility in distributed content-based publish/subscribe systems. In *Proceedings of the 2009 IEEE International Conference on Distributed Computing Systems (ICDCS 2009)*. Montreal, Canada.

Jacobsen, H.-A. The PADRES project. http://padres.msrg.utoronto.ca.

Kazemzadeh, R. S. and Jacobsen, H.-A. 2009. Reliable and highly available distributed publish/subscribe service. In *Proceedings of 2009 IEEE International Symposium on Reliable Distributed Systems (SRDS 2009)*. Niagara Falls, New York.

Kumar, V., Zhongtang, C., Cooper, B. F., Eisenhauer, G., Schwan, K., Mansour, M., Seshasayee, B., and Widener, P. 2006. Implementing diverse messaging models with self-managing properties using IFLOW. In *Proceedings of the 2006 IEEE International Conference on Autonomic Computing (ICAC 2006)*. Dublin, Ireland.

Li, G., Cheung, A., Hou, S., Hu, S., Muthusamy, V., Sherafat, R., Wun, A., Jacobsen, H.-A., and Manovski, S. 2007. Historic data access in publish/subscribe. In *Proceedings of the 2007 International Conference on Distributed Event-based Systems (DEBS 2007)*. Toronto, Canada.

Li, G. and Jacobsen, H.-A. 2005. Composite subscriptions in content-based publish/subscribe systems. In *Proceedings of the 2005 ACM/IFIP/USENIX International Conference on Middleware (Middleware 2005)*. Grenoble, France.

Li, G., Muthusamy, V., and Jacobsen, H.-A. 2008. Adaptive content-based routing in general overlay topologies. In *Proceedings of the 2008 ACM/IFIP/USENIX International Conference on Middleware (Middleware 2008)*. Leuven, Belgium.

Muth, P., Wodtke, D., Weisenfels, J., Dittrich, A. K., and Weikum, G. 1998. From centralized workflow specification to distributed workflow execution. *Journal of Intelligent Information Systems 10,* 2, 159–184.

Muthusamy, V., Jacobsen, H.-A., Chau, T., Chan, A., and Coulthard, P. 2009. SLA-driven business process management in SOA. In *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research (CASCON 2009)*. Toronto, Canada.

Nanda, M. G., Chandra, S., and Sarkar, V. 2004. Decentralizing execution of composite web services. In *Proceedings of the 2004 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004)*. Vancouver, Canada.

Pietzuch, P. R., Ledlie, J., Shneidman, J., Roussopoulos, M., Welsh, M., and Seltzer, M. I. 2006. Network-aware operator placement for stream-processing systems. In *Proceedings of the 2006 International Conference on Data Engineering (ICDE 2006)*. Atlanta, GA, USA.

White, S. 2004. Introduction to BPMN. http://www.bpmn.org/Documents/Introduction%20to%20BPMN.pdf.

Wodtke, D., Weissenfels, J., Weikum, G., and Dittrich, A. K. 1996. The mentor project: Steps toward enterprise-wide workflow management. In *Proceedings of the 1996 International Conference on Data Engineering (ICDE 1996)*. New Orleans, Louisiana.

Wun, A. and Jacobsen, H.-A. 2007. A policy management framework for content-based publish/subscribe middleware. In *Proceedings of the 2007 ACM/IFIP/USENIX International Conference on Middleware (Middleware 2007)*. Springer-Verlag New York, Inc., Newport Beach, California.

Yeung Cheung, A. K. and Jacobsen, H.-A. 2006. Dynamic load balancing in distributed content-based publish/subscribe. In *Proceedings of the 2006 ACM/IFIP/USENIX International Conference on Middleware (Middleware 2006)*. Melbourne, Australia.