FROM EVENT-DRIVEN WORKFLOWS TOWARDS *A POSTERIORI* COMPUTING

Craig A. Lee and B. Scott Michel
Computer Systems Research Department
The Aerospace Corporation, P.O. Box 92957
El Segundo, CA 90009
lee,scottm@aero.org

Ewa Deelman and Jim Blythe
USC/Information Sciences Institute
4676 Admiralty Way, Suite 1001
Marina del Rey, CA 90292
deelman,blythe@isi.edu

Abstract This chapter examines the integration of content-based event notification systems with workflow management. This is motivated by the need for dynamic, data-driven application systems which can dynamically discover, ingest data from, and interact with other application systems, including physical systems with online sensors and actuators. This requires workflows that can be dynamically reconfigured on-the-fly on the receipt of important events from both external physical systems and from other computational systems. Such a capability also supports fault tolerance, i.e., reconfiguring workflows on the receipt of failure events. When decentralized workflow management is considered, the need for a workflow agent framework becomes apparent. A key observation here is that systems providing truly autonomic, reconfigurable workflows cannot rely on any form of *a priori* knowledge, i.e., static information that is “compiled-in” to an application. Hence, applications will have to increasingly rely on *a posteriori* information that is discovered, understood, and ingested during run-time. In the most general case, this will require semantic analysis and planning to reach abstract goal states. These observations indicate that future generation grids could be pushed into more declarative programming methods and the use of artificial intelligence – topics that must be approached carefully to identify those problem domains where they can be successfully applied.

Keywords: Events, workflow, agent frameworks, artificial intelligence.

1. Introduction

This chapter examines the integration of content-based event notification systems with workflow management to enable dynamic, event-driven workflows. Such workflows are essential to enable a large class of applications that can dynamically and autonomously interact with their environment, including both computational and physical systems. The notion is that events delivered to a workflow underway may require the workflow to be redirected on-the-fly. Such events may signify external events in a physical system, such as pressure suddenly increasing in a chemical process, or it may signify failure in a computational resource. Considering this integration leads us to fundamental observations about future generation grids.

We begin by reviewing event notification systems, primarily within the context of event representation and distribution, but also discussing several currently available event systems. This is followed by a review of the elements of workflow management. Concrete realizations of these elements are described in the *Pegasus* system. We then consider event-driven workflows in the context of both centralized and decentralized management supported in a workflow agent framework. This leads us to observations concerning the reduced reliance on *a priori* information in favor of *a posteriori* information that is discovered during run-time. We conclude with a summary.

2. Event Notification Systems

Event notification is an essential part of any distributed system. Prior to briefly discussing several existing event notification systems, we first discuss *representation* and *distribution*; two fundamental concepts that define the capabilities of such systems.

2.1 Event Representation and Distribution

Representation expresses an event and its metadata that the distribution component propagates to interested receivers. Example representations, listed in order of expressiveness, include atoms, attribute-value pairs and interest regions. Atomic representation is generally useful for signaling simple events, such as start and completion of a processes, or heartbeats while a process is executing. Attribute-value representation signals and event and associated metadata, such as change in stock price or a simulated entity's position. Interest regions are generalized form of the attribute-value representation, where an event's metadata contains multiple attributes and associated values. An interest region is defined by the intersection between the values contained in the event's metadata and the values in which a potential receiver is interested; the event and its metadata are delivered to the receiver if the intersection is non-null.

The distribution component delivers an event, produced by a sender and signified by its representation, to interested receivers. In addition to propagating events from sender to receivers, the distribution component typically provides delivery semantics such as ordering, aggregation, reliable delivery, metadata filtering, receiver synchronization and interest management. Many of these delivery semantics are offered in combination with each others and depend on the the intended application environment. For example, a *topic-oriented publish/subscribe* system's distribution component generally offers temporally ordered, reliable event delivery represented as attribute-value pairs. Interest management in topic-oriented publish/subscribe event is implicit, since the receivers specify the events (topics) in which they are interested by subscribing to a sender (or senders) publishing the events. In contrast, a *content-oriented publish/subscribe* system offers richer capabilities since delivery is contingent upon examining an event's metadata before delivering the event to a receiver. Event representation expressiveness is thus tied to the distribution component's capabilities and the delivery semantics that it provides.

Before a distribution component can propagate events from senders (producers) to intended receivers (consumers), it must provide a rendezvous between the producers and consumers. The rendezvous problem's solution varies in sophistication and depends on the transport protocols available well as the relative sophistication of the distribution component and application needs. For example, a simple *"who-has"–"I-have"* protocol, based on a broadcast or multicast transport, could be used in a small scale environment with relatively few participants. In such a protocol, a consumer broadcasts a *"who-has"* message containing a topic name and waits for *"I-have"* responses from producers. The consumer subsequently registers its interest in receiving events with each producer from which a *"I-have"* response was received. A registry provides a more sophisticated rendezvous, where a producer registers its offered event topics with the registry. Consumers can either register their topics of interest with the registry or register their interest with actual producers. In the first case, the registry is responsible for propagating a producer's events to the consumers, whereas, in the second case, the producers propagate events directly to the consumers. In both cases, the registry is potentially a single point of failure, which tends to necessitate building a replicated registry infrastructure. Content-based event distribution requires a more sophisticated rendezvous, using either the broadcast protocol or a registry as a bootstrapping mechanism. The difficulty in content-based distribution is not the consumers discovering what topics are available, but informing the intermediate distribution components between the sender (producer) and the intended receivers (consumers) so that metadata processing and filtering can be performed according to the receiver's actual interests.

2.2 Current Event Notification Systems

These fundamental concepts for event representation and distribution define a host of implementation requirements that have been addressed by a number of currently available event notification systems. The CORBA Event Service [53], for instance, is built on the concept of an *event channel*. Event producers and consumers obtain an event channel by using a Naming Service or by opening a new channel. Both producers and consumers can be *push* or *pull-style*. That is to say, push producers asynchronously push events to push consumers. A pull producer waits until a pull consumer sends a request for events. If a push producer is on the same channel as a pull consumer, the channel must buffer pushed events until the consumer requests them. The Java Message Service (JMS) [49] provides a similar messaging capability where messages can be synchronous (pull) or asynchronous (push) using queues or publish-subscribe topics. The XCAT implementation of the Common Component Architecture (CCA) similarly provides a reliable network of message channels that supports push and pull for XML-based messages [26]. WS-Notification also has the notion of event publishers, consumers, topics, subscriptions, etc., for XML-based messages [43]. Regardless of the high-level interface that is presented, however, all such systems reduce to the fundamental issues of representation and distribution that have to be implemented with wide-area deployability and scalability.

In applications with a relatively small number of entities (i.e., tens of entities) participating in an event notification system, simple transport mechanisms such as broadcast or multicast transport suffice. Broadcast transport, if not supported by the underlying network stack and hardware, can be simulated by replicated unicast. Broadcasting and replicated unicast have the advantage of propagating events to all receivers, relying on the receiver to decide to accept or ignore the delivered event. As the number of participating entities increases, the burden on the underlying network increases, thus necessitating more sophisticated infrastructure. For example, the Scribe event notification system [42, 7] implements a topic-oriented publish/subscribe event distribution infrastructure whose aim is to provide efficient group communication. Scribe is implemented using the Pastry object location and routing infrastructure [41], a member of the distributed hash table (DHT) infrastructure family. Other content-oriented publish/subscribe event distribution systems have been proposed or built with the Chord infrastructure [48], e.g., [54, 47, 52]. Implementing publish/subscribe systems using DHT infrastructures can be challenging due to their origins as hash tables, which index a datum via its key. DHT keys are opaque binary identifiers, such as a SHA-1 hash [37] on a character string or the IP address of the participating node. Implementing a topic-oriented system is fairly straightforward: a hash on the topic becomes a DHT key. Content-oriented systems

hierarchically partition the DHT key, where each partition has some significance or predefined meaning. Hierarchical partitioning is necessary to take advantage of a DHT's efficient message routing, whether while updating a topic's content or delivering updated content to the topic's subscribers. If the partitioning is carefully designed, it is possible to use a DHT's message routing to evaluate range constraints on the event's data, as is demonstrated in [54].

Event distribution systems do not have to be artificially restricted to efficient message routing from publisher to subscribers or interested receivers. Another approach is to construct a system where the infrastructure itself operates on both the event and its metadata. In the *High Level Architecture* (HLA) [28] for federated simulation systems, interacting federates register interest in event interest regions based on the concept of *hyper-box intersection*: an event is delivered to a receiver if the event's attributes fall within the receiver's range of interests across a set of metrics. This can be implemented by exchanging aggregate interest maps among all federated end-hosts [6]. Another approach is to use active routing path nodes to perform interest management as part of an active network [27]. In the SANDS system [56], for instance, active router nodes inspect the entire event message flow to determine whether an event intersects a subscriber's interest region, and forward the message appropriately if the intersection was non-null.

The FLAPPS general-purpose peer-to-peer infrastructure toolkit [36] offers a hybrid approach between the efficient message passing structure of DHTs and the per-hop processing offered by active networks. The FLAPPS toolkit is designed for store-and-forward message passing between peers communicating over an *overlay network*. Messages are forwarded from a source peer through one or more transit peers en route to the receiver(s). This store-and-forward approach allows each transit peer to operate on the message's contents and decide whether to continue forwarding the message onward. In an event distribution system, FLAPPS enables the transit peers to operate on the event messages. FLAPPS is designed with extensibility in mind, so that different types of overlay network and different styles of message passing are enabled. It is possible to create an overlay network and message passing system that emulates Chord or Scribe/Pastry. Similarly, it is possible to emulate active networking if the overlay network mimics the underlying Internet's structure. FLAPPS enables content-aware, application-level internetworking between overlay networks, such that events from an event distribution system propagating in one overlay network can be mapped to events in another event distribution in its own overlay network, and vice versa. Alternatively, an overlay network could represent the distribution tree for one attribute in an event's metadata such that the message forwarding between overlay networks has the effect of performing interest region intersection [34]. While this inter-overlay interest region message forwarding primitive has not yet been built as a FLAPPS service, it

is a demonstrative example what is possible given a sufficiently flexible and general-purpose communication infrastructure.

The use of overlay networks for event notification, and messaging in general, is a very powerful concept for grid environments that can be used in many contexts. As an example, NaradaBrokering [39, 21] is a distributed messaging infrastructure that facilitates communication between entities and also provides an event notification framework for routing messages from producers only to registered consumers. All communication is asynchronous and all messages are considered to be events that can denote data interchange, transactions, method invocations, system conditions, etc. The topology of overlay networks can also be exploited to provide a number of *topology-aware communication services* as argued in [33, 32]. These services includes relatively simple communication functions, such as filtering, compression, encryption, etc., collective operations, but also content-based and policy-based routing, in addition to managed communication scopes in the same sense as MPI communicators. Hence, the power of overlay networks and their range of applicability make them very attractive for distributed event notification systems.

3. The Elements of Workflow Management

Workflows help formalize the computational process. They allow users to describe their applications in a sequence of discrete tasks and define the dependencies between them. To a scientist, a workflow could represent a desired scientific analysis. To an engineer, a workflow could represent the computational processes that control a physical process, such as a chemical plant. In the most general sense, a workflow could represent the flow of information among the sensors, computational processes, and actuators, that comprise a dynamic, event-driven system.

Initially, workflows are typically built or planned as *abstract workflows* without regard to the resources needed to instantiate them. To actually execute a workflow in a distributed environment, one needs to identify the resources necessary to execute the workflow tasks and to identify the data needed by them. Once these resources (compute and data) are identified, the workflow must be *mapped* on to the resources. Finally, a *workflow execution engine* can execute the workflow defined by the abstract definition. Although users may be able to locate the resources and data and construct executable workflows to some extent, the resulting workflows are often inflexible and prone to failures due to user error as well to changes in the execution environment, e.g., faults.

In the following, we discuss these elements of workflow management in more detail. We then describe how they are implemented in the *Pegasus* workflow management system [10, 12, 14]. Several grid applications using Pegasus are subsequently discussed briefly to illustrate the breadth of its applicability.

3.1 Workflow Representation and Planning

An abstract workflow is essentially a representation of a workflow. This representation can be *built* or *planned* in a variety of ways.

The simplest technique is appropriate for application developers who are comfortable with the notions of workflows and have experience in designing executable workflows (workflows already tied to a particular set of resources.) They may choose to design the abstract workflows directly according to a predefined schema. Another method uses Chimera [17] to build the abstract workflow based on the user-provided partial, logical workflow descriptions specified in Chimera's Virtual Data Language (VDL). In a third method, abstract workflows may also be constructed using assistance from intelligent workflow editors such as the Composition Analysis Tool (CAT) [31]. CAT uses formal planning techniques and ontologies to support flexible mixed-initiative workflow composition that can critique partial workflows composed by users and offer suggestions to fix composition errors and to complete the workflow templates. Workflow templates are in a sense skeletons that identify the necessary computational steps and their order but do not include the input data. These methods of constructing the abstract workflow can be viewed as appropriate for different circumstances and user backgrounds, from those very familiar with the details of the execution environment to those that wish to reason solely at the application level.

Examining the workflow construction tools from a broader perspective, the tools should also be able to cope with changing user goals, in addition to a dynamic grid environment. Hence, such tools cannot rely on solutions that are pre-programmed in advance but must instead construct and allocate workflows based on information about the users needs and the current state of the environment. A variety of approaches may be appropriate for this task, including automated reasoning, constraint satisfaction and scheduling. Artificial Intelligence (AI) planning and scheduling techniques have been used because there is a natural correspondence between workflows and plans. Planning can be used in several ways in a general workflow management system. For example, it has been used to construct abstract and concrete workflows from component specifications and a list of desired user data [12] and planning concepts have been used to help users construct abstract workflows interactively [31]. In all cases, the grid workflow construction problem can be cast as a planning problem, which we now describe.

AI planning systems aim to construct a directed, acyclic graph (DAG) of actions that can be applied to an initial situation to satisfy some goal. Classical planning systems take three kinds of input: (1) a description of the initial state of the world in some formal language, (2) a description of the agent's goals, which can be tested against a world state, and (3) a set of operators. The operators are parameterized actions that the planning system can apply to transform a state

into a new one. Each operator is described by a set of preconditions that must hold in any state in which the operator is to be applied, and a set of effects that describe the changes to the world that result from applying the operator. Implemented planning systems have used a wide range of search techniques to construct a DAG of actions to satisfy their goals, and have been applied in many practical applications [24].

To cast grid workflow management as a planning problem, we model each application component as an operator whose effects and preconditions come from two information sources: the hardware and software resource requirements of the component and the data dependencies between its inputs and outputs. The planner's goal is a request for a specific file or piece of information by the user. If the file does not yet exist, it will be created by composing the software components in the resulting plan. The operator definitions ensure that the plan will be executable, even if some of the planned components also have input file requirements that are not currently available and require auxiliary components. The planner's initial state includes the relevant files that already exist and their locations, hardware resources available to the user and, when available, bandwidth and resource time estimates. While it is possible for the planner to make queries to these services during planning, it is often not practical due to the large amount of queries they generate.

This general framework can be adapted to a variety of situations with different levels of available information, metadata or modeling of components or resources [3]. For example, the number of operators required can be greatly reduced by using metadata descriptions with a formal language to describe the contents of a file, since one operator can describe many situations where a component may be used, each of which may require a separate operator definition if files are viewed as atomic objects without metadata descriptions. Similarly, by using metadata to describe a group of files, for example those involved in a parameter sweep, the planner can efficiently reason about the group as a unit, distinguishing its members when needed because of resources or file availability. Using these techniques we have used a planner to generate workflows involving tens of thousands of jobs very quickly. On the other hand, the planner can still be used if this information is not available, or if resource information is not available. In this case, an abstract workflow is produced, rather than one that both selects components and allocates them to resources.

Although the planner can in principle create concrete workflows before they are executed, in practice it is often useful to separate the stages of abstract workflow construction and resource allocation because the dynamic nature of grid applications makes it more appropriate to allocate resources closer to the time they are to be used. The use of explicit representations and algorithms for workflow planning and scheduling helps us to formalize and explore the trade-off of allocating ahead of time or allocating close to execution.

Most planning systems allow users to add constraints on the plans that will be produced, even when planning is not interactive. Often, these constraints are required for plans to be found efficiently. In our work we have used search control rules to direct the planner to plans that are likely to be efficient, to prune parts of the search space that cannot lead to good plans, and to ensure that the planner quickly samples the space of available plans when asked to generate several possibilities. Search control rules constrain or order the available options at specific choice points in the planning algorithm, for example to choose or eliminate the application component that is used to provide a required file, or to consider a particular kind of resource first before considering other resources. Some of the rules we use are applicable in any grid problem while some are only applicable to a particular set of application components or information goals.

3.2 Resource Selection and Workflow Mapping

Regardless of the methods employed in designing the initial workflow, we view the abstract workflow as a form of intermediate representation that provides the basis for mapping the workflow onto the necessary resources. The workflow mapping problem can then be defined as finding a mapping of tasks to resources that minimizes the overall workflow execution time, which is determined by the running time of the computational tasks and the data transfer tasks that stage data in and out of the computations. In general, this mapping problem is NP-hard, so heuristics must be used to guide the search towards a solution.

First, the workflow mapping system needs to examine the execution environment and determine what are the available resources and whether the workflow can be executed using them. Since the execution environment is dynamic the availability of the resources and thus the feasibility of the workflow execution can vary over time. Later on in this section, we look at how these dynamic changes can be adapted to by the mapping system (via workflow partitioning for example).

The execution environment is rich with information provided by a variety of information services. Some of the information refers to the computational resources such as the set of available resources, their characteristics (load, job queue length, job submission servers, data transfer servers, etc.) Some of the information refers to the location of the data referenced in the workflow. In distributed environments, it is often beneficial to replicate the data both for performance and reliability reasons, thus the data location services may return several locations for the same data set. Obviously, one also needs to take into account the existence and location of the software necessary to execute the analysis described in the workflow. That information along with the environment

that needs to be set up for the software, any libraries that need to be present, etc. can also be provided by the execution environment.

Given this information, the mapping needs to consider which resources to use to execute the tasks in the workflow as well as from where to access the data. These two issues are inter-related because the choice of execution site may depend on the location of the data and the data site selection may depend on where the computation will take place. If the data sets involved in the computation are large, one may favor moving the computation close to the data. On the other hand if the computation is significant compared to the cost of data transfer the compute site selection could be considered first.

The choice of execution location is complex and involves taking into account two main issues: feasibility and performance. Feasibility determines whether a particular resource is suitable for execution, for example whether the user can authenticate to the resource over the period of time necessary for execution, whether the software is available at the resource and whether the resource characteristics match the needs of the application. The set of feasible resources can then be considered by the mapping system as set of potential candidates for the execution of the workflow components. The mapping system can then choose the resources that will result in an efficient execution of the overall workflow.

There are several issues that affect the overall performance of a workflow defined as the end-to-end runtime of the workflow. In compute-intensive workflows, the decision where to conduct the computations may be of paramount importance. One may want to use high-performance resources to achieve the desired performance. Additionally, as part of the resource allocation problem one may also need to consider parallel applications and their special needs such as how many processors and what type of network interconnect are needed to obtain good performance. One complexity that also often arises in distributed environments is that resources can be shared among many users. As part of the compute resource selection problem, one may also want to consider whether to stage-in the application software (if feasible) or whether to use pre-installed applications. In data-intensive workflows that are often seen in scientific applications, it may be more beneficial to consider data access and data placement as the guiding decisions in resource selection. The choice of which data to access may depend on the bandwidths between the data source and the execution site, the performance of the storage system and the performance of the destination data transfer server.

Also, it is possible that the intermediate (or even final) data products defined in the workflow have already been computed and are available for access. As a result it may be more efficient to simply access the data rather then recomputed it. However, if the data is stored on a tape storage system with slow access, it may be desirable to derive it again.

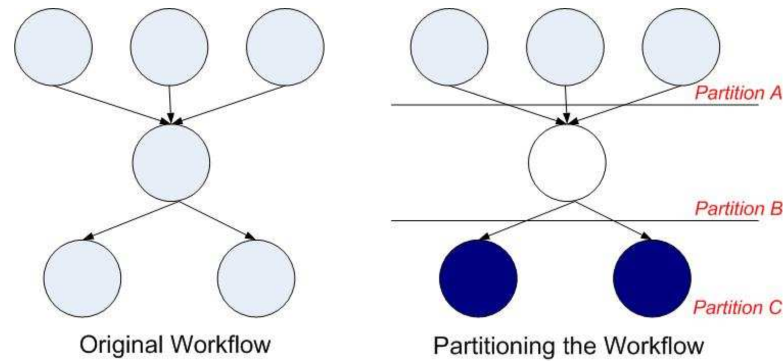

Figure 1. An example of workflow partitioning based on the level of the nodes in the workflows.

In general, one can only estimate the performance of the workflow components on a variety of resources and the time it takes to transfer the data across the network. One severe complication is that oftentimes the size of the intermediate data that is accessed and generated during the execution of the workflow is hard to estimate. Since both the data transfer and the runtime of the workflow components are affected by the data size, it may be hard to estimate the end-to-end workflow performance a priori.

As we have already noted, the execution environment is very dynamic with both the resource availability and the resource characteristics varying over time. At the same time, scientific workflows are often large, consisting of thousands or hundreds of thousands of individual tasks. The combination of these two factors and the uncertainties in the runtime of the components and data sizes makes it very hard to estimate the end-to-end workflow performance behavior ahead of time. One may think that it would sufficient to map the workflow to the resources as the workflow components become ready to run. However, doing so may result in poor performance. It may be due to unnecessary data transfer between components, or to the overloading of critical resources early in the workflow making them unavailable later on, etc. One can alleviate contention for resources or lack of resources by reserving resources ahead of time-bringing us back to the mapping of the entire workflow ahead of time and reserving the resources a priori.

The best solution seems to lie somewhere between the mapping of the entire workflow and mapping portions of the workflow. This brings up the key concept of *partitioning* workflows into subworkflows that maintain the dependencies among the original workflow components, thus simplifying the overall workflow structure. Figure 1 shows an example partitioning that divides the workflow based on the level of the components in the workflow.

At this point, one can plan out the entire workflow but submit only the portions of the workflow that can be run in the near future. We can denote

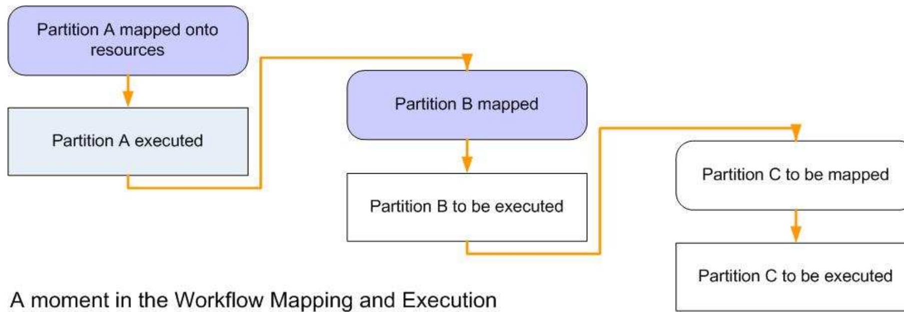

Figure 2. An example of workflow mapping and execution for a workflow composed of 3 linearly-dependent partitions.

how far into the future to release the workflow as the *scheduling horizon*. As the execution progresses and the execution environment changes, the initial workflow mapping may need to be adjusted. Mapping the entire workflow ahead of the execution may be very costly and not appropriate for cases when the execution environment changes rapidly. In such cases it may be beneficial to derive a *mapping horizon* indicating how far into the future (how far into the workflow) to map the tasks. As the workflow is executed, the workflow horizon is increased further and the mapping of the resulting portions of the workflow is being conducted. Figure 2 shows an example of this workflow refinement and incremental execution process for the workflow partitioned as in Figure 1.

In general we can imagine that one can dynamically set the mapping and scheduling horizons based on the workflow characteristics (size and number of tasks) and the behavior of the execution environment. Additionally these horizons may differ, with a greater mapping horizon allowing for a possibly better overall schedule and the shorter scheduling horizon improving the execution time of the workflow. For example if the target execution system is quite static over time and the application workflow is small, then it may be most efficient to have the entire workflow encapsulated by one partition. On the other hand if the execution environment is very dynamic, it may be beneficial to divide the workflow so that each node of the workflow forms a single partition that is mapped and scheduled accordingly.

Finally, we wish to point out that the mapping and scheduling process is itself a workflow, or rather a *meta-workflow*, that is determined by application requirements.

3.3 Workflow Execution

Once the abstract workflow is mapped onto the available resources, resulting in an executable workflow, it can be given to an execution system for execution. One of the widely used systems is DAGMan [23]. Given a DAG in a specific format, DAGMan follows the dependencies between the tasks to determine the set of task ready for execution. The ready tasks are those whose "parent" tasks (if any) have successfully completed. These tasks are released to Condor-G for execution on remote resources or to Condor, if the resources are local. Upon completion of tasks new ready tasks are determined. DAGMan provide logging facilities that collect information about the completion status of the tasks, which Condor maintains the status of the tasks submitted for execution. Additionally, DAGMan provides fault tolerance and fault handling capabilities. For example when a task represented in a DAG node fails, DAGMan can retry it a certain number of times. If a task ultimately fails, DAGMan stops the workflow execution and returns a *rescue DAG* to the user. This DAG contains all the nodes that are left to be executed including the failed tasks. The user or a workflow management system may then modify the rescue DAG and resubmit it for execution.

DAGMan is one of the basic workflow execution engines, it contains the basic mechanism for execution and fault tolerance. Other systems have been developed such as GridAnt [2]that targets solely the grid environment. GridAnt extends the Ant (http://ant.apache.org) Java build tool, which manages dependencies in the project build process, to produce a workflow processing engine, which can manage task dependencies is a distributed environment. Many other workflow engines are embedded into workflow management or creation systems that provide higher-level functionality. For example systems such as Triana [51], Kepler [1] and Taverna [38] enable users to graphically construct workflows, often case executable workflows, and then execute them in a variety of environments such as peer-to-peer networks and grids.

3.4 Workflow Management in Pegasus

Many of the workflow concepts have been embodied in a system called Pegasus [10, 12, 14], which stands for Planning for Execution in Grids. Pegasus is a framework that maps complex scientific workflows onto distributed resources such as the Grid [19]. From the point-of-view of the user, Pegasus can run workflows across multiple heterogeneous resources distributed in the wide area, while at the same time shielding the user from the details of the distributed environment. From the point-of-view of performance, there are great benefits to the Pegasus approach to application description, mapping, and execution. The workflow exposes the structure of the application and its maximum parallelism. Pegasus can then take advantage of the structure to set the mapping horizon to

adjust to the volatility of the target execution system. This feature is beneficial both in cases where resources or data may become suddenly unavailable and in cases where new resources come online. In the latter case, Pegasus can opportunistically take advantage of these newly available resources. The exposure of the maximum parallelism also enables Pegasus to cluster tasks together to reduce the overheads of target scheduling systems.

In Pegasus, workflow composition is modeled as a *planning problem* [3]. Each application component is considered to be a *planning operator*. A partial order is imposed on the operators that observes their input and output data dependencies. Any required data movement, such as a file transfer across the network, is also modeled as a planning operator. The heart of the Pegasus workflow planner uses AI planning techniques that allow a declarative representation of the workflow components. The Prodigy planning architecture [55] is used with *search control rules* [25] to build the plan along with learning modules designed to reduce the planning time and improve the quality of the solution plans.

Pegasus assumes that the environment provides services to locate the resources and provide their characteristics. The actual site selection is a pluggable component within Pegasus but several standard selection algorithms are available [4]. In case of data resources, Pegasus also uses services and catalogs that provide information about the location of data referenced in the workflow. Pegasus consults catalogs that provide information about the location of the application software and its requirements. When using the CAT software, an input data selector component can use a Metadata Catalog Service (MCS) [44, 13] and a Replica Location Service (RLS) [8] to populate the workflow template with the necessary data.

Since the space of possible workflow partitions and schedules is large, Pegasus implements a basic horizon setting mechanism where the scheduling horizon is equal to the mapping horizon. In our initial implementation the mapping horizon is set statically based on the structure of the workflow. Pegasus provides capabilities to partition workflows but users can provide their own function.

Pegasus and DAGMan has been used to successfully execute both large workflows with an order of 100,000 jobs with relatively short runtimes and workflows with small number of long-running jobs. Pegasus and DAGMan were able to map and execute workflows on a variety of platforms: condor pools, clusters managed by LSF or PBS, TeraGrid hosts (www.teragrid.org), and individual hosts.

Pegasus has been used for a number of significant applications. Pegasus was used to manage a large number of small jobs for Montage [29], a grid-capable astronomical mosaicing application. Pegasus was similarly used for a galaxy morphology application [15]. ANL scientists used Pegasus to achieve a

speedup of 5-20 times on the BLAST bioinformatics application by efficiently keeping the submission of jobs to a cluster computer constant. To contrast, Pegasus was used to manage a few, long-running jobs for high-energy physics applications such as Atlas and CMS [12]. The workflows required by the Laser Interferometer Gravitational-Wave Observatory (LIGO) to detect gravitational waves emitted by pulsars are characterized by many medium and small jobs [11, 16]. Recently, Pegasus has been used in demonstration of earthquake analysis where three different seismic hazard-related calculations were created and executed using the SCEC portal (Southern California Earthquake Center www.scec.org) in a secure manner. These applications clearly demonstrate the robust workflow capabilities of Pegasus.

4. Event-Driven Workflows

While statically configured and executed workflows will be completely sufficient for many applications, the notion of *dynamic, event-driven workflows* will enable a fundamentally different class of applications. In this class of applications, events that occur after the workflow has been started can potentially alter the workflow and what it is intended to accomplish. We may wish to change a workflow for a variety of reasons, and the events that trigger these changes may come from any source possible. Workflow-altering events may occur in external, physical systems, in the application processes themselves, or in the computational infrastructure. For example, a workflow manager running a pollution dispersion code suite may get an event signifying that a particular chemical species has been detected, thus requiring a particular physics module in the code suite to be turned on. As an alternative example, a workflow manager running a distributed Monte Carlo simulation may get an event signifying that a host running a segment of the simulation has failed or become unreachable, thus requiring the segment to be restarted on a different host. Clearly the ability to dynamically manage workflows is an important capability that will enable dynamic application systems and also improved reliability.

4.1 Workflow Control

To understand what event-driven workflows require, we must examine and understand what event delivery means to a workflow manager. In general terms, this can mean

- Understanding what the event means in the context of the current application,
- Understanding the goals of current workflow,
- Determining what the new goals of the workflow should be, i.e., *replanning the workflow*,

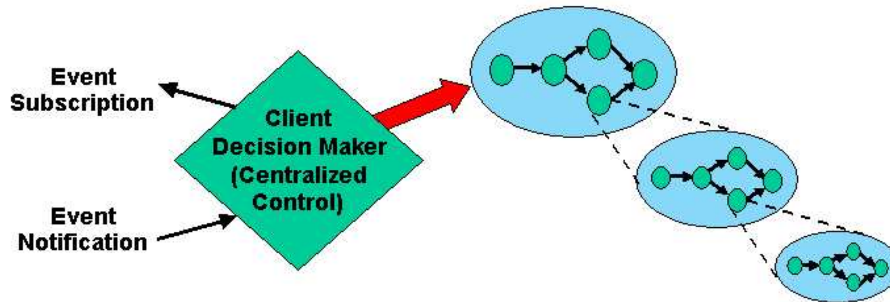

Figure 3. Centralized Workflow Management.

- Understanding the current state of the workflow, and
- Determining the exact logistics of converting the current workflow to the new workflow.

Clearly a workflow manager needs to have complete control over an existing workflow as a prerequisite to enabling an altered or new workflow. How this complete control is exerted depending greatly on whether the workflow management is centralized (orchestrated) or decentralized (choreographed).

Figure 3 illustrates a centralized workflow enactment engines. This centralized engine understands the goals of the workflow and has global knowledge of (at least) the top-level services in the workflow. This centralized engine also represents one place where events concerning the workflow must be delivered. Up on receipt of a relevant event, the engine may decide to (1) do nothing, (2) cancel part of the workflow, or (3) cancel the entire workflow. Regardless of whether the engine has co-scheduled the entire workflow or incrementally scheduled specific services, the engine must know which services are currently active (executing), and which services are not yet active. Planned services that are not yet active can simply be discarded. (If the resources were co-scheduled, they can be released.) For services that are active, the engine can (a) synchronously wait for them to complete, or (b) asynchronously terminate them. If the currently active services are allowed to complete, then the engine can assume that valid output data sets are available on the service hosts. These can be used in any replanned workflow. If the currently active services are prematurely terminated, then the engine cannot assume that any output data sets are valid. In either case, the disposition of any data sets on the service hosts must be taken care of to eliminate any unused "stale" data sets or references to data sets that do not exist.

Currently active services would be terminated by the use of *cancellation events* sent to the services initiated by the engine. We note, however, that if

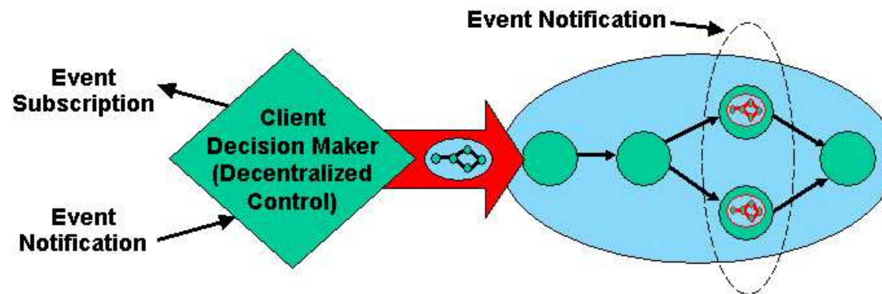

Figure 4. Decentralized Workflow Management.

workflow services are *nested* or *recursive*, then even a centralized engine may not know the entire workflow state. In this case, we could rely on the service call tree to propagate the cancellation events. Alternatively nested or recursive service calls could listen on a specific *cancellation event topic*, effectively allowing the service call tree to be "short-circuited".

While centralized workflow engines will also be completely sufficient for many applications, we note that it does represent a single point of failure. Hence, as Figure 4 illustrates, the decision to initiate a workflow and the run-time management of the workflow are separated and, in fact, a representation of the workflow is passed among the workflow service hosts allowing the workflow management to be *decentralized*. Each service host in the workflow understands where it sits in the entire workflow. While all of the workflow hosts could be co-scheduled, it would also be possible for each service host to incrementally schedule the following services. We note that nested or recursive workflows are likewise possible here.

In this scenario, however, there is no one place where relevant events can be delivered to alter the workflow. Clearly, when the workflow is initiated, each service host, whenever it is scheduled, must be able to subscribe to a specific event topic whereby it will receive all relevant events. Likewise, the decision criteria for what the event means and what to do about it must also be propagated among the service hosts. At any particular time when an event occurs, it needs to be delivered to the currently active service hosts. We note that there could be *event delivery skew* among these hosts. Nonetheless, the currently active service hosts should be aware of each other and be able to synchronize appropriately. In addition, the currently active service hosts must agree on the meaning of the event and also be able to *collaboratively replan* the workflow.

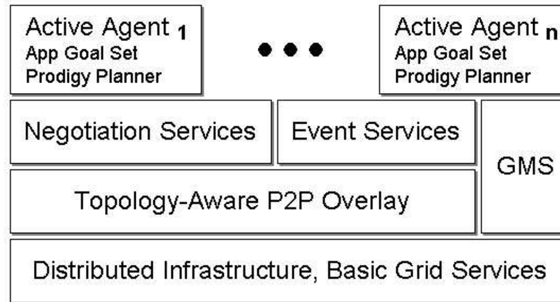

Figure 5. A Workflow Agent Framework.

4.2 A Workflow Agent Framework

The consideration of decentralized workflow management prompts the notion of workflow services as *workflow agents*. Besides just providing a specific service, each service host can also act as an agent by deciding conditional workflow branches, deciding subsequent service scheduling, receiving events and potentially replanning the workflow. The consideration of workflow agents motivates the need for a *workflow agent framework* as illustrated in Figure 5. Such a framework would provide the fundamental services needed for agents to interact by managing the flow of control and data, of all types, among service hosts.

This framework will clearly be built on top of a distributed infrastructure of basic grid services and network communication. On top of these base services, however, will be a peer-to-peer, overlay network that is topology-aware. The size and topology of this P2P overlay can be managed such that enhanced communication services can be provided as noted in [33, 32]. One such communication service is clearly an *event service* that supports the appropriate flow-based management with both *topic-based* and *content-based* event distribution. Agents will be able to produce and listen for specific event types and content to proper manage workflow processes.

Workflow management in an agent framework means that agents would have to potentially collaborate on each step of the workflow life cycle, e.g., planning (or replanning), resource selection, workflow mapping, and finally workflow execution. As noted, if an agent receives an event that indicates a workflow modification, then the current state of the system and the ultimate goal of the workflow may both change. In this case, replanning could involve running a distributed version of Prodigy where all agents participate in arriving at the same new plan based on the current set of application goals. Alternatively, each agent may formulate their own plan and then *negotiate* with their peers to agree

on the best course of action. Hence, the framework should also support a set of message-oriented *negotiation services*.

For those agent workflows that are expected to exhibit a high degree of dynamic behavior, the notions of a *scheduling horizon* and a *mapping horizon* gain even more significance. Allowing a workflow to be incrementally scheduled and mapped allows the trade-off to be made between cooperatively planning ahead versus reacting to a dynamic environment. The "distance" to the horizon, i.e., how much of the workflow can be planned, co-scheduled, and mapped as a unit, will depend on how volatile the environment is expected to be, and also where key choice points may occur within the workflow, that is to say, where alternative courses of action may be more likely. Hence, the negotiation services will have to support not only the initial workflow structuring, but also the partitioning, mapping and scheduling horizon functions.

The notion of agents collaborating on workflows is strongly related to a number of existing fields. Workflow management actually builds on the notion of *process programming* [50]. Process programming languages have the goals of process composibility with the appropriate level of abstractions, semantic richness to support multiple paradigms, and ease of use. If such tools are to be used in a distributed environment, we can also consider *agent coordination languages* [45, 35]. In these environments, processes or agents coordinate via messages that can represent processing. These messages may be explicitly addressed, point-to-point messages, or be associatively addressed and delivered via a *tuple space*. This notion of *active messages* can be generalized to *active agents*. Hence, we can see that the P2P overlay network is actually supporting a rich messaging service capable of supporting event distribution, negotiation, and other communication-oriented services.

While it would make life easy to assume that workflow agents will operate in a reliable environment, this will certainly not be the case. Hence, the framework should incorporate some *grid monitoring service* (GMS) that can produce fault events as appropriate, e.g., when an agent fails or cannot communicate with its collaborators. While fault detection is essential, we will also want workflow agents to exhibit *autonomic behavior*. At the most general level, autonomic computing in intended to be "self-defining, self-configuring, self-optimizing, self-protecting, and self-healing". To realize these behaviors on a practical level requires that an agent must be able to accomplish precisely the behaviors required for dynamic workflow management, i.e., (1) detect and understand the failure event, (2) understand the ultimate goals of the workflow, and (3) be able to replan, reschedule, and remap the workflow. This must be done even though the agent may have imperfect knowledge of the environment and limited control over it [18]. Under these conditions, we must realize that many dynamic workflows may only be capable of approximately accomplishing their goals.

5. Towards *A Posteriori* Computing

This examination of dynamic, event-driven workflows is actually an instance of a much larger issue in grid computing. The term "grid" essentially denotes an open-ended, heterogeneous computing infrastructure where many, if not all, aspects of a particular computation or application can be dynamically decided or adapted. In fact, grid applications that are typically considered to be "more interesting" are precisely those that have a high degree of capability, flexibility, or robustness that is not possible any other way. Grids promise to support *virtual organizations* of people and resources that can be dynamically organized on-the-fly for specific goals. Rather than statically configuring their applications, grid users may expect their applications to discover and use suitable resources at run-time. Likewise, grid users may expect their applications to autonomically recover many types of failures and carry-on to successful completion. As we concluded in the examination of robust, decentralized workflow management, agents should be able to understand the goals of an application, replan, and initiate the steps necessary to achieve the goal state.

Hence, this ultimate potential of grids will be more fully realized as more things are dynamically decided and supported at run-time. In decades past, on single, isolated machines, the local operating system was able to completely manage all resource discovery and usage by an application. These resources were disk drives, tape drives, printers, etc. Where an application ran was not an option but the operating system could manage many tasks at one time. When an application ran, it was by-and-large statically built and configured with a significant amount of *a priori* information concerning its environment and how it could be used. In this context, *a priori* information essentially means *knowable independently of experience*. This is information or knowledge that is "compiled-in" or acquired "out-of-band". It is static since it expects the world to have certain properties and be in a known state. Since it is static and known ahead of time, semantic translation tools (i.e., compilers) can be used to analyze and optimize entire code units. In short, everything that can be statically defined *a priori* removes complexity from the application and can improve performance.

Grid systems, middleware, and applications, on the other hand, will increasingly rely on *a posteriori* information – knowledge that is learned from experience. While this may the increase complexity and degrade the performance of grid systems, this also enables exactly the kind of flexibility that is expected from grids. We briefly examine the sources of flexibility and dynamic behavior in grids.

5.1 Sources of Dynamic Behavior

Besides events that are application domain-specific, the sources of dynamic behavior in grids are many. The goal of many grid applications is to dynamically discover suitable resources at start-time. Such discovery depends on the persistent information services that are available, e.g., the Globus MDS, etc. As we have noted many times, the subsequent resource selection, scheduling, and mapping functions may be very dynamic. Advance reservation tools such as GARA [20] may be useful but only up to any scheduling horizon.

The notion of resource discovery and selection can be generalized to that of service discovery and selection. When considering a web services architecture, dynamic behavior will be derived from dynamic Web Service Definition Language (WSDL) interpretation. (In current implementations, WSDL is statically interpreted to generate a proxy class prior to application compile-time.)

Middleware that is able to discover an application's environment and make decisions concerning the dynamic scheduling of resources and the composition of services could be considered a "smart run-time" library, or even a "smart back-end" of the compiler. The GrADS project (Grid Application Development System), for example, provides a Runtime Compiling System that dynamically composes services while collecting performance information that is used in the Scheduler/Service Negotiator and also feedback to the program preparation environment [30].

The concept of middleware being used to specifically capture dynamic information about an application is far-reaching. Such middleware will be able to capture, for example, how an application was built, where it is executing, its communication patterns, data requirements, execution times, etc. Such captured dynamic information could be used for a smart back-end but could also be used for reliability purposes. The MPICH-V system, for example, provides a message-logging infrastructure that essentially watches the program execution to enable transparent fault tolerance [5].

Such dynamic information is critical to supporting the *Autonomic Control Cycle: monitor, analyze, plan, execute*. But this only serves to illustrate the difficulty in providing such capabilities in non-trivial ways. In the old Globus Heartbeat Monitor [46], for instance, a monitor process would listen for heartbeat messages from running applications. The meaning of a late heartbeat was highly ambiguous. Was the missing heartbeat caused by a process failure, a host failure, or a network failure; or was the heartbeat simply late? It is impossible to disambiguate the situation without gathering more information and *inferring* a probable cause. In the case of the Heartbeat Monitor, this would entail one monitor process asking another if it also missed a heartbeat from the same host or along the same network path. Such capabilities will have to be developed to achieve autonomic computing for robust workflows and intelligent agents,

where failures can even be predictively anticipated and humans can be removed from the loop.

5.2 What Must Still Be "Compiled-In"?

Even though grids offer much greater flexibility both in the environment and in applications, there is still much common knowledge that must be "compiled in". Such common knowledge entails base-level services for the most basic capabilities – not only for the discovery and composition of resources and services, but also for the discovery of the *representation* of these services and their associated data, events, behaviors, and semantics. This implies a fundamental, or well-known, ontology that can bridge metadata schemas. For any collaborative workflows among agents, this will have to include representations for goals and the associated planning and inferencing to reach them. Such knowledge must be discoverable through a simple, well-known mechanism.

In the field of Artificial Intelligence, the ultimate notion of *a posteriori* means that absolutely all knowledge must be gained from experience. In this extreme case, an agent would essentially be a *tabula rasa* that must learn everything from the environment through interactions that are seemingly random and non-goal directed. It is certainly useful to understand the properties and bounds of all possible intelligent systems but all practical systems will have some amount of knowledge built-in. For example, any host in an open-ended, distributed environment would have to have some type of network connectivity and understand a communication protocol such as TCP/IP to participate in the environment, even if it did not know to discover, compose, and use services at a higher layer. Hence, in the ultimate analysis, a grid agent will come closest to *a posteriori* computing when it is built on a *provably minimal set of well-known behaviors* through which it can discover the rest of its grid world.

6. A Summary of Future Grids

This chapter has examined the integration of content-based event notification systems with workflow management tools to support dynamic, event-driven workflows. The key issue that this examination has emphasized is that supporting decentralized workflow management is tantamount to supporting a grid agent framework. While agents may inhabit this framework to achieve a workflow goal, their dynamic nature requires that they must be able to subscribe to topic-based or content-based event distribution in order to interact with the larger environment.

Another important observation is that to achieve flexibility and autonomy, we will need to "compile-in" less *a priori* information into grid workflow agents and to require them to discover more dynamic information about their environment, i.e., to acquire *a posteriori* information. The fact that less "rote knowledge" will

be compiled-in means that agents will be pushed into using more declarative programming methods and that more artificial intelligence techniques will be adopted. That is to say, agents will be programmed with the "what" rather than the "how". Such declarative programming techniques will support inferencing to allow autonomous agents to proceed from their current state to a goal state. The metadata schemas and ontologies that would enable such workflow agents to function in a grid environment are strongly related to those required for the "Semantic Grid" [40].

Finally we note that dynamic, event-driven workflows are also highly relevant to Dynamic, Data-Driven Application Systems [9]. These are application systems that can dynamically discover and ingest new data from their environment and then respond appropriately. This class of applications includes, for example, using high resolution radars to spatially resolve and track tornadoes, ingesting ground data to drive climate, combustion, and turbulence models related to forest fires, and monitoring and controlling heavy road traffic. Such applications, using dynamic, event-driven workflows, will ultimately be part of a much larger *cyberinfrastructure* [22].

Acknowledgments

This work is supported by NSF OISE-0334238, US-France (INRIA) Cooperative Research: Active Middleware Grids, and by the Aerospace Parallel Computing Mission-Oriented Investigation and Experimentation (MOIE) project. Pegasus is supported by NSF under grants ITR-0086044 (GriPhyN), ITR AST0122449 (NVO) and EAR-0122464 (SCEC/ITR). Pegasus was developed at USC/ISI. In addition to Ewa Deelman, the Pegasus team consists of Gaurang Mehta, Gurmeet Singh, Mei-Hui Su, and Karan Vahi.

References

[1] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludäscher, and S. Mock:. Kepler: An extensible system for design and execution of scientific workflows. In *16th International Conference on Scientific and Statistical Database Management*, 2004.

[2] K. Amin et al. GridAnt: A Client-Controllable Grid Workflow System. In *37th Annual Hawaii International Conference on System Sciences*, January 2004.

[3] J. Blythe, E. Deelman, and Y. Gil. Automatically composed workflows for grid environments. *IEEE Intelligent Systems*, 19(4):16–23, July/August 2004.

[4] J. Blythe et al. Task scheduling strategies for workflow-based applications in grids. In *International Symposium on Cluster Computing and the Grid*, 2005. To appear.

[5] G. Bosilca et al. MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes. In *IEEE/ACM Supercomputing '02*, November 2002.

[6] S. Brunett, D. Davis, T. Gottschalk, P. Messina, and C. Kesselman. Implementing distributed synthetic forces simulations in metacomputing environments. In *Proceedings of the Heterogeneous Computing Workshop*, March 1998.

[7] M. Castro, P. Druschel, A. M. Kermarrec, and A. I. T. Rowstron. Scribe: a large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*, 20(8):1489–99, 2002.

[8] A. Chervenak et al. Giggle: a framework for constructing scalable replica location services. In *Supercomputing*, 2002.

[9] F. Darema. Grid Computing and Beyond: The Context of Dynamic, Data-Driven Application Systems. *Proceedings of the IEEE*, March 2005. Special issue on Grid Computing, M. Parashar and C. Lee, guest editors.

[10] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi, and M. Livny. Pegasus : Mapping scientific workflows onto the grid. In *Across Grids Conference 2004*, Nicosia, Cyprus, 2004.

[11] E. Deelman et al. *GriPhyN and LIGO, Building a Virtual Data Grid for Gravitational Wave Scientists.* in 11th Intl Symposium on High Performance Distributed Computing. 2002., 2002.

[12] E. Deelman et al. Mapping abstract complex workflows onto grid environments. *Journal of Grid Computing, 2003.*, 1(1), 2003.

[13] E. Deelman et al. Grid-based metadata services. In *16th International Conference on Scientific and Statistical Database Management*, 2004.

[14] Ewa Deelman, James Blythe, Yolanda Gil, and Carl Kesselman. *The Grid Resource Management*, chapter Workflow Management in GriPhyN. Kluwer, 2003.

[15] Ewa Deelman et al. Grid-based galaxy morphology analysis for the national virtual observatory. In *SC 2003*, 2003.

[16] Ewa Deelman et al. Pegasus and the pulsar search: From metadata to execution on the grid. In *Applications Grid Workshop, PPAM 2003*, Czestochowa, Poland, 2003.

[17] I. Foster et al. *Chimera: A Virtual Data System for Representing, Querying, and Automating Data Derivation.* in Scientific and Statistical Database Management. 2002., 2002.

[18] I. Foster, N. Jennings, and C. Kesselman. Brain Meets Brawn: Why Grid and Agents Need Each Other. In *AAMAS*, July 19-23 2004.

[19] I. Foster, C. Kesselman, and eds. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann Publishers, Inc., 1998., 1998.

[20] I. Foster, C. Kesselman, C. Lee, B. Lindell, K. Nahrstedt, and A. Roy. A distributed resource management architecture that supports advance rservations and co-allocation. In *7th IEEE/IFIP International Workshop on Quality of Service*, 1999.

[21] G. Fox and S. Pallickara. An Event Service to Support Grid Computational Environments. *Concurrency and Computation: Practice & Experience*, 14(13-15):1097–1129, 2002.

[22] P. Freeman et al. Cyberinfrastructure for Science and Engineering: Promises and Challenges. *Proceedings of the IEEE*, March 2005. Special issue on Grid Computing, M. Parashar and C. Lee, guest editors.

[23] J. Frey, T. Tannenbaum, M. Livny, and S. Tuecke. *Condor-G: A Computation Managament Agent for Multi-Institutional Grids.* In Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC10), San Francisco, California., 2001.

[24] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann Publishers, 2004.

[25] Y. Gil, E. Deelman, J. Blythe, C. Kesselman, and H. Tangmunarunkit. Artificial Intelligence and Grids: Workflow Planning and Beyond. *IEEE Intelligent Systems*, pages 26–33, Jan.-Feb. 2004.

[26] M. Govindaraju et al. Merging the CCA Component Model with the OGSI Framework. In *International Symposium on Cluster Computing and the Grid*, May 2003.

[27] Active Networks Working Group. Architectural framework for active networks. http://www.cc.gatech.edu/projects/canes/arch/arch-0-9.ps, 1999.

[28] HLA Working Group. IEEE standard for modeling and simulation (M&S) high level architecture (HLA) - framework and rules. IEEE Standard 1516-2000, 2000.

[29] J. C. Jacob et al. The montage architecture for grid-enabled science processing of large, distributed datasets. In *Proceedings of the Earth Science Technology Conference (ESTC)*, 2004.

[30] Ken Kennedy et al. Toward a framework for preparing and executing adaptive grid programs. In *Proceedings of NSF Next Generation Systems Program Workshop (International Parallel and Distributed Processing Symposium)*, April 2002.

[31] J. Kim et al. A knowledge-based approach to interactive workflow composition. In *Proceedings of Workshop: Planning and Scheduling for Web and Grid Services at the 14th International Conference on Automatic Planning and Scheduling (ICAPS 04)*, Whistler, Canada, 2004.

[32] C. Lee. Topology-aware communication in wide-area message-passing. In *Recent Advances in Parallel Virtual Machines and Message Passing Interface*, pages 644–652, 2003. Springer-Verlag LNCS 2840.

[33] C. Lee, E. Coe, B.S. Michel, I. Solis, J. Stepanek, J.M. Clark, and B. Davis. Using topology-aware communication services in grid environments. In *Workshop on Grids and Advanced Networks, International Symposium on Cluster Computing and the Grid*, May 2003.

[34] C. Lee and S. Michel. The use of content-based routing to support events, coordination and topology-aware communication in wide-area grid environments. In D. Marinescu and C. A. Lee, editors, *Process Coordination and Ubiquitous Computing*, pages 99–118. CRC Press, 2003.

[35] C. Lee and D. Talia. Grid programming models: Current tools, issues and directions. In Berman, Fox, and Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*, pages 555–578. Wiley, 2003.

[36] B. Scott Michel. *General-purpose Peer-to-Peer Infrastructure*. PhD thesis, University of California, Los Angeles, 2004.

[37] NIST. The secure hash algorithm (sha-1). Technical Report NIST FIPS PUB 180-1, National Institute of Standards and Technology, U.S. Department of Commerce, April 1995.

[38] T. Oinn et al. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics Journal*, 20(17):3045–3054, 2004.

[39] S. Pallickara and G. Fox. NaradaBrokering: A Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids. In *ACM/IFIP/USENIX International Middleware Conference*, pages 41 – 61, 2003.

[40] D. De Roure, N. Jennings, and N. Shadbolt. The Semantic Grid: Past, Present and Future. *Proceedings of the IEEE*, March 2005. Special issue on Grid Computing, M. Parashar and C. Lee, guest editors.

[41] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM Intl Conf. on Distributed Systems Platforms (Middleware 2001)*, November 2001. Heidelberg, Germany.

[42] A. Rowstron, A-M. Kermarrec, M. Castro, and P. Druschel. Scribe: The design of a large-scale event notification infrastructure. In *Network Group Communication (NGC 2001)*, London, UK, 2001.

[43] S. Graham and others. Publish-Subscribe Notification for Web Services. http://www-106.ibm.com/developerworks/library/ws-pubsub, 2004.

[44] G. Singh et al. *A Metadata Catalog Service for Data Intensive Applications*. SC03, 2003.

[45] D. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2), June 1998.

[46] P. Stelling, C. DeMatteis, I. Foster, C. Kesselman, C. Lee, and G. von Laszewski. A fault detection service for wide area distributed computations. *Cluster Computing*, 2(2):117–128, 1999. Special Issue on HPDC-7.

[47] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. *ACM. Computer Communication Review*, 32:73–86, 2002.

[48] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM Computer Communication Review*, 31(4):149–160, 2001.

[49] SUN Microsystems, Inc. Java Message Service API. http://java.sun.com/products/jms, 2002.

[50] S.M. Sutton and L.J. Osterweil. The design of the next generation process language. In *Joint 6th European Software Engineering Conference and the 5th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 141–158, 1997.

[51] I. Taylor, I. Wang, M. Shields, and S. Majithia. Distributed computing with Triana on the Grid. *Concurrency and Computation:Practice and Experience*, 17(1–18), 2005.

[52] Wesley W. Terpstra, Stefan Behnel, Ludger Fiege, Andreas Zeidler, and Alejandro P. Buchmann. A peer-to-peer approach to content-based publish/subscribe. In *International Workshop on Distributed Event Based Systems (DEBS03)*, 2003.

[53] The Object Management Group. CORBA 3 Release Information. *http://www.omg.org/technology/corba/ corba3releaseinfo.htm*, 2000.

[54] P. Triantafillou and I. Aekaterinidis. Content-based publish/subscribe systems over structured p2p networks. In *International Workshop on Distributed Event Based Systems (DEBS04)*, May 2004.

[55] M. Veloso et al. Integrating Planning and Learning: The PRODIGY Architecture. *J. of Experimental and Theoretical Artificial Intelligence*, 7(1), 1995.

[56] S. Zabele et al. SANDS: Specialized Active Networking for Distributed Simulation. *DARPA Active Network Conference and Exposition*, pages 356–365, May 29-30 2002.