USING SOFTWARE DESIGN PATTERNS TO BUILD DISTRIBUTED ENVIRONMENTAL MONITORING APPLICATIONS[1]

DAG JOHANSEN, KJETIL JACOBSEN, NILS P. SUDMANN, KÅRE J. LAUVSET, KENNETH P. BIRMAN AND WERNER VOGELS

Tools developers face the challenge of exposing a development methodology to users while concealing details of the underlying system. If that system is complex and subject to evolution, this problem can be particularly difficult. Here, we discuss the use of software design patterns in conjunction with StormCast, a system of tools developed to support environmental and weather monitoring tasks in the Arctic. Now entering its 5th generation, each version of StormCast has expanded the capabilities of the underlying distributed data management tools and computational facilities. This paper reviews StormCast 5.0, presents the design patterns used by developers, and describes several applications in terms of the application of these patterns.

[1] The first authors were supported by the Norwegian Research Council (NFR), grant No. 17543/410 and 112578/431 (DITS program). Birman and Vogels were supported in part by DARPA ITO under ARPA/ONR contract N0014-96-1-10014 and ARPA/RADC contract F30602-95-1-0047, and in part by grants from Siemens, Intel and Microsoft.

INTRODUCTION

Developers of software tools that will serve broad communities often view software engineering as a means of guiding application developers towards good architectures. This paper describes the use of *software design patterns* for this purpose. Software design patterns are well known to the software engineering community, but there has been little experience in applying this approach to large distributed computing systems. In so doing, our work validated the methodology, but also required some extensions to the most commonly used design patterns.

Our effort arose out of a distributed data management system called StormCast [Hartsvigsen and Johansen 1990][Johansen and Hartvigsen 1995]. Developed to support environment and weather monitoring, StormCast aggregates data collection, archiving, processing and display resources distributed over Norway and the adjacent seas. Data sources include land-based data repositories, sensors located on remote islands and sea-going trawlers, satellite data feeds, and other information sources. StormCast has several processing centers, and end-users access the system from dozens of locations, including over the Web. StormCast includes a variety of data treatment applications and can support several "styles" of weather application, each requiring its own development methodology. Moreover, StormCast has evolved steadily over a decade, with each new version exposing significant new functionality to users.

StormCast brings together three communities with very different goals. The authors of this paper were involved in the team that developed the internal architecture of StormCast and the distributed computing techniques used to implement the system. StormCast offers a set of "tools" to a second community, members of whom use it to develop weather and environment monitoring application programs. The third community operates these applications to collect and process incoming sensory data, yielding weather and environmental reports for diverse uses.

Our challenge is that application programmers for this domain are rarely experts in distributed computing. Like any system, StormCast is far more effective in supporting certain styles of applications than others. Moreover, StormCast is increasingly complex: We are now completing the 5th version of the overall system, and each new generation of the system has typically been able to support the applications that ran on previous versions while also enabling new applications that exploit different styles of communication or computation. The application developer thus faces a number of choices, and may be ill-prepared to make them. These choices have significant implications for the quality of the solution. An application that uses StormCast inefficiently may incur high overheads or perform costly, unnecessary computations.

We are using software design patterns to address this problem. Each pattern represents a style of application, providing development guidance to application builders who face applications of the corresponding type. The developer works by identifying one or more design patterns that match the goals and structure of a target application. When a good match is found, the developer refines it into useable piece of software with some confidence that the resulting application will be well supported by StormCast and will continue to work well as the system evolves.

The problem we face is a common one. Many distributed systems are of a scale and complexity that demands some form of application design support. Application developers who will work over such systems are likely to have their hands full merely providing domain-specific functionality. They may not have the time or interest required to learn how the underlying platform was structured, or to

understand the reason that one style of solution will outperform another, operate more reliably, or achieve a higher level of security. Our success in solving similar problems using software design patterns in StormCast suggests that the methodology is a viable response to this issue.

The body of this paper is structured as follows. The next section discusses StormCast itself in somewhat more detail. The section following it introduces software design patterns, and then discusses the major design patterns relevant to StormCast. Three sidebars describe some major StormCast applications and their experiences using the method. The paper concludes with a summary of our findings.

STORMCAST 5.0

StormCast is a collection of wide-area weather and environmental monitoring applications that operate in "realistic" environments: Heterogeneous networks, with varying and unpredictable connectivity, user demands, numbers of users, and occurrences of failures. StormCast's role is primarily pedagogical: our objective is to understand how to build applications of this sort, but not to develop and operate a "production quality" environmental monitoring system. We seek to show application builders and future users how the tools and ideas of the distributed computing research community can be exploited to solve their real-world problems.

StormCast can be understood as an underlying "system" over which a number of applications operate, sharing resources and data that the system manages. Application development has been done in cooperation with domain specialists, such as meteorologists and pollution experts, current and potential end-users of this technology, and industrial collaborators. Jointly, we have prototyped a number of distributed applications for monitoring, collecting, processing and visualizing this type of data and information. Examples include weather monitoring systems for an airport, a system for monitoring pollution parameters in a river, one for monitoring air pollution in a valley, and a mobile platform for monitoring and collecting data on board of fishing trawlers and other vessels.

The core StormCast system is thus concerned with the automation of meteorological and environmental data collection, storage, and analysis. This is done by providing tools that support user-developed applications, a typical example being one that predicts upcoming local and global weather in a correct and timely way. This application has to trigger alarms whenever a dramatic change is to occur or has just occurred, present the (vast) associated data set in a customized and reduced form, and archive summary data and processed information for future usage. To work correctly, the application must spread functionality over a heterogeneous wide-area network. This entails providing timely delivery of data even when network partitionings are frequent, placing computations that reduce data volume close to data archives, and dealing with issues of scale. Moreover, scaling occurs in several dimensions: Such an application may need to accommodate large numbers of end-users, may need to be extensible and customizable for specialized uses, and may

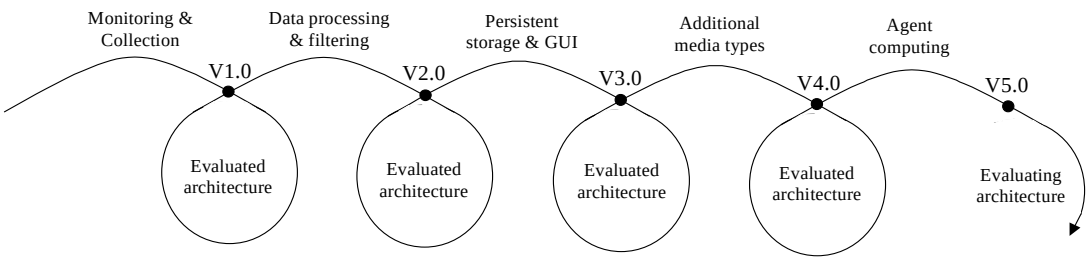

Figure 1: The StormCast domain architecture has evolved incrementally over several years.

need to filter small amounts of useful information out of huge amounts of data.

The StormCast architecture has been evaluated and iterated over a number of implementations. Each iteration was used to implement a set of associated applications, evaluated in terms of performance and flexibility, and then refined to obtain a new version of the architecture. Over time, an increasingly mature and powerful architecture has been derived. Figure 1 illustrates this process, where an initial model guided the first architecture. Figure 2 illustrates how the different components are mapped into the StormCast 5.0 architecture. We have tended to associate version numbers with major new kinds of functionality, as indicated on the labeled arcs in Figure 1.

StormCast 5.0, the most recent member of this series, runs on a WWW platform with the core part of the application located at the University of Tromsø. StormCast handles multiple media types and integrates a diverse set of services from its own and external service providers. For example, the Tromsø subsystem monitors, collects (polls) and stores raw weather data from weather sensors located on the University roof. It also grabs and stores a video picture from a still camera every 15 minutes throughout the year. An additional video camera can be controlled by remote users[2].

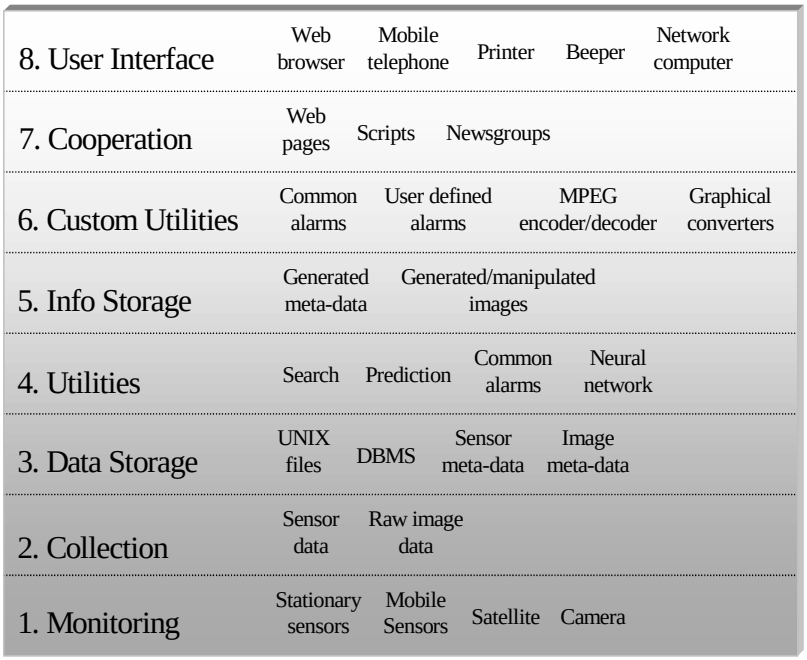

Figure 2: StormCast 5.0 architecture and its components.

StormCast also integrates a number of initially disjoint components developed outside of our effort, creating what appears to the user as a single, albeit loosely coupled, distributed computing environment. For instance, satellite images are provided daily from the Tromsø Satellite Station. The Norwegian Meteorological Institute has also provided us with weather prediction data for the Northern Europe based on a weather model (LAM 50S) run on a remote mainframe. We have prototyped a back-end volume visualizer which converts 120 Mbyte weather predictions into 3D images. This StormView application is further detailed in sidebar 3.

The requirement for persistent storage has changed over the years. In the early versions of StormCast, flat Unix files were used to store immutable, time-stamped weather data. With the introduction of additional and diverse media types, we needed a more flexible storage system. Hence, StormCast 5.0 uses a (Write-Once, Maybe-Read) common storage system implemented

[2] Try it out by pointing your web browser to URL: http://www.cs.uit.no/DOS/StormCast

through PostgreSQL and standard Unix files. PostgreSQL primarily contains meta-data which is associated with the respective raw data (Binary Large Objects - BLOBs) found in the Unix file(s). This coupling is given through Uniform Resource Locators (URL) where HTTP acts as a transport mechanism. This duality was primarily motivated by scaling needs; our approach facilitates the storing of BLOBs on multiple (remote) hosts with large disks. When we replicate BLOBs for reliability and availability purposes, we also mask locality (provide location and replication transparency) through this extra meta-level of indirection.

It is undesirable to have many users or higher layer services repeatedly perform identical data reductions, so the utility layer in StormCast 5.0 contains "abstract" utility services that compute data as needed and cache data likely to be reused. These services operate by transforming data obtained from the bottom layers into higher-level forms of information. Implicit here is a reduction in data volume. Examples of utilities range from tools querying the storage services to software retrieving data associated with URLs. An instance of the former is a client-side library supplied with the storage system. With this library, a client program written in Perl, C or C++ can send arbitrary SQL commands to the database. This is especially useful for non-interactive queries. An instance of the latter service in StormCast 5.0 is one that does a date–to-date search for certain image types such as, for instance, satellite or panoramic weather images. The result is given as one or multiple URLs, and the user then fetches relevant images using a utility program for display on a web-browser.

We can take advantage of the SQL part of the storage system to make complex searches. For instance, in StormCast 5.0 a user can search for satellite images containing meta-data attributes. Thus, a StormCast satellite image might be classified as 'cloudy', 'stormy' or 'clear' in the (longitude-latitude) specified regions.

The task of the application developer involves customizing the existing remote StormCast core services and adding software to them. This can be done in two fundamentally different ways.

First, application developers can build client-side StormCast software running *locally* in their browser environments. Early StormCast versions used X11 and Windows for this functionality. At that time, we found it hard to add users dynamically and very resource intensive to customize any part of the GUI. Today, however, StormCast uses a GUI based on HTML, and this has proved to be much simpler. Customized front-end applications are now linked to the remote core StormCast services by means of their respective URLs. A very simple example from StormCast 5.0 is a customized HTML page containing the fundamental StormCast URLs for current weather predictions.

A more elaborate example is an HTML page with status snapshots periodically pushed from the various services. Such a page can be constructed by looking at the HTML source for the various services and pick the items of interest. Given this information, the user can construct a web page that displays, for instance, the latest panorama picture, the latest weather status and the latest satellite image. A user can also build specialized animations of, for instance, satellite images with clear weather by searching the database for satellite images with the corresponding meta-data. Sidebar 1 also details how this meta-tagging can be automated.

The second option for StormCast application developers is to install their application components *remotely*, closest to the core StormCast servers. One way of doing this is to install a server or server extension manually, and then use RPC for communication with this software. This does not scale to very large user communities, however, so StormCast 5.0 also supports an alternative and very flexible solution based on *agent* technology. Agents are highly specialized services automating some of the special needs for an individual user or small user group [Riecken 94].

We are, in particular, exploring *mobile* agent techniques in StormCast using a subsystem we call TACOMA [Johansen 95b, Johansen 96]. TACOMA is an operating system supporting network agents implemented in C, Tcl/Tk, Perl, Scheme, Python, or OCaml. A notable feature of TACOMA is that it can launch an agent from a web-browser, something that makes StormCast easily scalable. Through a web browser, an application programmer can now install customized StormCast agent software remotely and potentially save network bandwidth and/or network messages. This type of agent does not have to be moved after installation (although it may need to be *removed* on the basis of StormCast's management policy). Agents travel over the network to collect or produce necessary information for the user, and then return to the user carrying the results of the computation.

To illustrate this type of agent functionality, consider a user-defined weather alarm. The application developer creates the agent by interacting with a local program that generates a personal agent for monitoring of certain weather parameters. For instance, the user might be interested in an alarm being activated when the wind speed exceeds 20 m/s and the temperature is below 0 degrees Celsius, or when a pressure sensor detects a sudden drop in air pressure. Next, this agent is exported to the remote server through an HTML form and CGI-script technology. From now on, the user is only notified when a certain event is detected (i.e. the alarm is triggered). The user can be notified and alerted through electronic mail or GSM phones.

SOFTWARE DESIGN PATTERNS

The diversity of the StormCast architecture and application base creates a challenging software engineering problem. Although the system is capable of supporting a great variety of weather applications, only certain styles of applications can be expected to perform well and to use system resources efficiently. Our challenge is to guide the developer to a good solution without working one-on-one with each potential user of the StormCast software base. Moreover, this should occur in a way that encourages the reuse of existing code where possible.

In early versions of StormCast we approached this problem by interacting directly with application developers. The application developer would tell us enough about the application to enable us to point to some existing application with a similar structure, or to identify a match with StormCast's existing mechanisms and utilities. But as the system has matured, it is clear that this sort of one-on-one dialog is less and less appropriate. The need is for an architectural framework that would permit

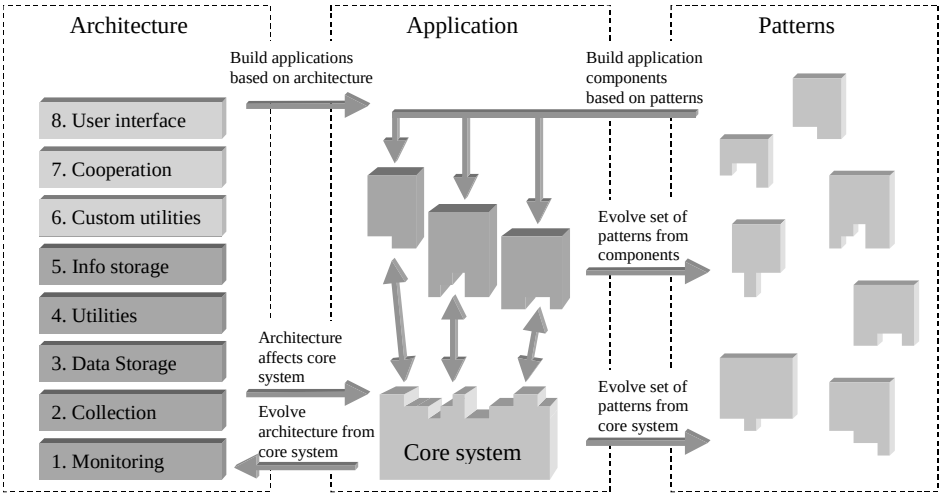

Figure 3: This is how the architecture, the application and the patterns are related. The whole process is an incremental one, where the application, the architecture and the patterns are revised as the project progress.

the designer to approach StormCast not just as an integrated set of technologies and data management tools, but rather in terms of design patterns that guide the developer towards successful patterns of thinking and styles of applications.

The patterns thus reflect the architecture of the system but do so implicitly, shielding the user from direct knowledge of how StormCast really operates. Figure 3 illustrates the relationship between the StormCast 5.0 architecture (on the left), a typical StormCast application (middle) and the design patterns used to implement it (right).

It should be stressed that we are approaching software design patterns as a "conceptual" rather than a completely practical methodological tool. That is, unlike some software engineering approaches that correspond directly to tools (such as GUI builders and systems for implementing database clients), our concern is that the developer be shown the styles of application that StormCast can efficiently support in a manner that facilitates identifying the best match for a proposed new application. Thus, at present, the design patterns are descriptive. This said, however, we view the development of "wizards" similar to the ones used on PC operating systems such as NT as an appealing way to automate the use of design patterns. (Users who have programmed under NT will be familiar with its many tools for stamping out applications in template form, which can then be refined by the developer who wishes to introduce application-specific functionality or to overload some standard function with a modified version).

Our use of design patterns for this purpose is not, in and of itself, particularly novel: this approach has gained a considerable following over recent years [Aarsten 1996] [Garlan 1995] [Boasson 1995] [Reicken 1996] [Scmidth 1996]. Indeed, many common patterns have been identified and are widely used within the software engineering community, and some of these have also been applied in StormCast, like the Reactor pattern, the Observer pattern and the State pattern, to name a few [Gamma 1995]. However, through our work, we have also identified less known patterns, like, for instance, the Alarm pattern, the Scripting pattern and the Agent pattern. These patterns will be detailed in the following subsections, which we structure in a pattern language style motivated by [Scmidth 1996] and [Gamma 1995]. The components of this style are self-evident; for brevity we omit a detailed discussion.

THE ALARM PATTERN

PROBLEM

A major use of StormCast involves monitoring some condition passively, so as to trigger an appropriate response if a sufficiently important condition arises. Examples include the detection of potentially severe weather patterns, detection of a hardware or communications failure that will require human service, or detection of an internal system problem, such as a shortage of space on a storage device. The goal of the alarm pattern is to assist the

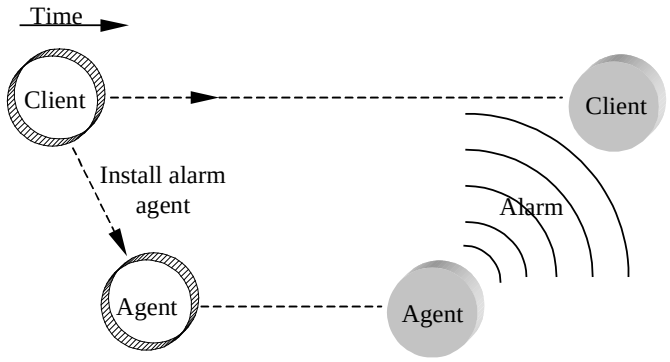

Figure 4: A client installs an alarm. The client is later notified when an invariant is violated.

application developer in programming StormCast to implement such alarms, thereby avoiding the need for a human operator to continuously monitor the underlying data source. A well-programmed alarm should notify the operator only if intervention is required.

Alarm patterns can be contrasted with another pattern that we discuss momentarily: the "publish/subscribe" pattern, whereby certain classes of events are communicated in real-time to a set of display or listening programs. Alarms trigger actions only when a state invariant is violated, or (in the case of more complex alarms), when a series of state transitions occurs.

StormCast is capable of supporting two kinds of alarms. "Generic" alarms are those in which a potentially large number of users or subscribers may need to be informed if the alarm triggers. "Specialized" alarms are typically associated with just a single user, often a system administrator, the weather specialist in a designated airport, etc.

ALSO KNOWN AS

Monitor, Failure Detector

CONTEXT

Alarms operate in the context of the data collection, storage and utilities layers of StormCast, which are physically decentralized and handle very large quantities of data. For example, a single satellite image may have a 20Mbyte representation. A consequence is that alarms may need to do their computing remote from the user who will be notified if the alarm condition arises. Moreover, StormCast's support for alarms must be extremely flexible, because the types of alarms active in the system have evolved as new types of sensors are introduced and as new utility processing has been defined.

FORCES

Alarms must accommodate a great variety of available data, complex detection rules, and notifications to a user community that (at least in the generic case) can vary while the system is running. It follows that alarms must be both *flexible* and *reconfigurable*. On the other hand, these features must not prevent efficient implementation of alarms, because the computational cost of monitoring large volumes of data is potentially significant. Some alarms have real-time constraints, such as those that monitor for sudden changes in weather conditions. Indeed, most weather related alarms have a real-time element. Since the notification target is a human, real-time response times on the order of seconds are typical objectives.

SOLUTIONS

There are many ways to implement alarms in StormCast. For brevity, we limit ourselves to the most common style of alarm, in which the application can be represented as an agent program, which operates to detect the alarm condition, and a notification message, which is sent to the end-user. Generic notifications are transmitted using publish/subscribe mechanisms (described in a separate pattern), email messages to sets of clients, or even cellular phone messages. The lists of users to notify and types of notification are configured, for each generic alarm, through a web page maintained on behalf of that alarm.

Specialized alarms are also modeled as agents, but whereas the user normally just needs to add their name to a list in the case of a generic alarm, the specialized ones are typically programmed on a per-need basis. In some cases, this can be automated using a high-level GUI that generates the necessary agent code. Otherwise, if the developer must implement a completely new agent, this is done using the StormCast subsystem called TACOMA, as described in Section 2.

EXAMPLES

Norway's proximity to several large and potentially unsafe nuclear power plants has made radioactivity levels a potentially important parameter to monitor. In this case, we have learned that using a simple threshold value leads to excessive rates of false notifications, because of natural fluctuations. The StormCast application for this domain currently implements two alarms: a basic alarm that monitors medium and long-term background radiation levels, notifying the user if a slow rise exceeding some threshold occurs, and a second superimposed alarm that monitors for sudden spikes of higher radiation. Both alarms are implemented in a decentralized manner: one copy is sent to each storage location at which raw radiation measurements are archived.

RELATIONSHIPS

This pattern makes use of the agent and the publish/subscribe patterns.

THE ASSEMBLY LINE PATTERN

PROBLEM

Much of the raw data managed by StormCast takes the form of time-series, such as periodic sensor measurements or periodic satellite images. Additional data sources are continuous generators of sensor data, which may be in any of a number of media types. All of this data is spread within a decentralized system. Yet end-users will often need reports that combine information from multiple servers. A related problem involves computations that have a pipelined structure, with each of a

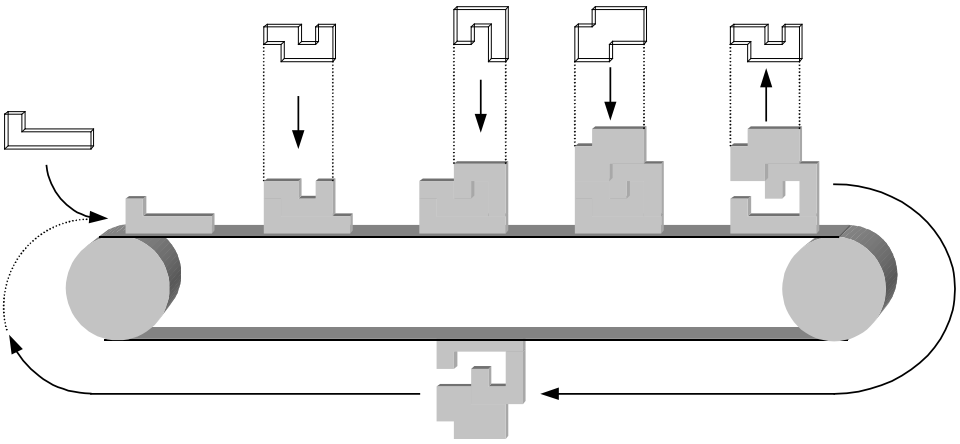

Figure 5: Assembly line pattern pattern

series of hosts performing some computation on the input from an upstream host.

ALSO KNOWN AS

Computer supported cooperative work (CSCW), Server Push, Work-Flow, Worm Programming, Active Mail.

CONTEXT

The StormCast architecture involves no single data repository at which all information is archived. Thus, many computations that, viewed in the abstract, operate upon the StormCast data set as a whole must be implemented by performing multiple sub-computations, each of which runs on a different server. The assembly line pattern arises in cases where such a computation can be structured into a sequence of processing elements, similar to a "pipe" in the UNIX operating system.

FORCES

The cost of gathering all the data needed for a computation can be prohibitive in the StormCast architecture, hence the assembly line pattern is intended to encourage users to consider parallel processing methods that substantially reduce the overall volume of data in each stage of computation. StormCast can support more than one such pattern (see the Controller/Worker pattern, below); the Assembly Line pattern covers the case where only one process is involved at a time. Some assembly line computations are iterative, in the sense that each occurrence of some type of event triggers a new execution of the pipelined computation.

IMPLEMENTATION

Much like the Alarm Pattern, the Assembly Line Pattern is well matched to StormCast's programmable agent facility. TACOMA supports agent mobility, and permits each mobile agent to carry data with it. Thus, the Assembly Line is typically implemented by handing the data item (or each data item, in the iterative case) to an agent, which travels from processor to processor requesting local computation at each and carrying the refined output to the next stage. The TACOMA agent language is flexible enough so that agents can be programmed to dynamically adapt their behavior on the basis of the results obtained at each stage, for example by skipping certain processing sites or adding additional ones.

EXAMPLE

When creating a weather forecast, a meteorologist starts with local measurements, satellite images of local relevance, and computer generated predictions. However, to obtain adequate detail for incoming fronts or other mid-range and long-range factors, it may be necessary to visit other servers, and from these, still others. The Assembly Line Pattern captures this behavior in the general case. Similarly, a severe weather advisory associated with a very small weather system may result in the generation of an agent that travels from server to server to inform each of the event, the predicted behavior (and level of confidence) for which being updated at each stage. Eventually, as the confidence drops, the associated agent will terminate. Yet a third example arises when a forecast must be reviewed or modified by a series of experts, before finally being published for use by the general user community.

RELATIONSHIPS

Agent pattern.

THE PUBLISH/SUBSCRIBE PATTERN

PROBLEM

Despite its decentralized structure, it is convenient for the developer to be able to treat StormCast as it possessed a single shared communication fabric. Information publishers should be able to make information available without prior knowledge of the membership of the set of receivers, and without needing to know where receivers are located. Information consumers should be able to retrieve published information without needing to know which processes are publishing it or where they are located, and without coordinating with other consumers.

CONTEXT

The StormCast system is implemented as a loosely coupled collection of servers distributed over a WAN, and connected to sensors by links of varying characteristics. Server-to-server links have non-uniform properties and may be disrupted by occasional partitioning events. Yet the application developer must be isolated from these properties, much as in the Internet News subsystem. In distinction to Internet News, data is published from process to process (not human to human) and may be produced at extremely high speeds. Moreover, News gives only best-effort reliability guarantees and performs garbage collection in an uncontrolled manner. StormCast application developers need guarantees of reliability and data consistency, and must control the persistence of critical data objects and ensure that transient information will be garbage collected quickly, to reclaim any space it might otherwise occupy.

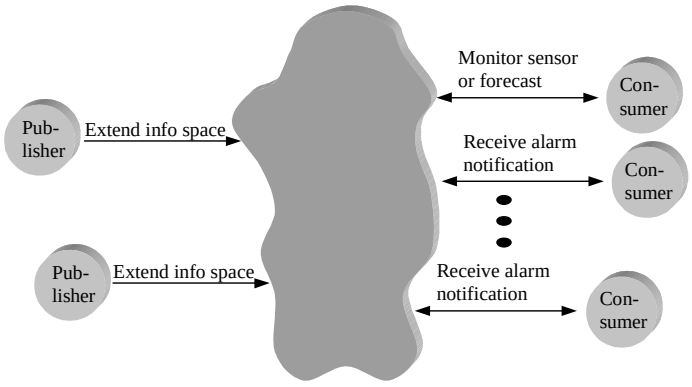

Figure 6: Published information is placed into an information space and is organized by "subject". Consumers receive copies of the data on subjects to which they subscribe. The reliability and ordering properties of the model are controlled by the application developer.

FORCES

There must be a large "name space" for the various types of data produced, architected to allow easy extension as new applications are developed. The number of producers will often be small, but they may be scattered throughout the network, and new producers for a given type of data may appear suddenly and need to be able begin publishing data with minimal overhead. The number of subscribers may be very large, and is similarly dynamic. Subscribers should not interfere with one another, and should receive data with minimal delays and with the degree of reliability guarantee

needed by the application. Control of the "properties" of the publication system should be possible on a per-publication basis.

SOLUTION

StormCast offers several mechanisms that solve this problem in varied settings [Birman 1993, Birman 1997]. Although we have indicated that for many uses, Internet News (or WWW publishing through a "push" server) will be unsatisfactory, these are both supported options. WWW publishing of a passive type can also be employed, although in this case the user's browser must repeatedly poll for updates. The Isis Toolkit was used in developing StormCast, and it includes a program to program publish/subscribe technology called MDS (Message Distribution Service) which scales to large numbers of processes while guaranteeing fault-tolerance and flow control. The performance of the approach is very good; to get a sense of its capacity, it is interesting to note that Isis MDS is also a key element of the New York and Swiss Electronic Stock Exchanges. Both use it to continuously publish every stock trade that occurs in these exchanges, and have subscriber populations numbered in the hundreds.

EXAMPLES

As part of its local weather service, StormCast publishes a number of weather sensor values and predictions, which can be "tapped" by applications that use this information or display it for human operators. The publishing part of this application consists of a customized service placed within StormCast's utility layer. This specific publishing system operates by repeatedly modifying a web page, which provides a brief formatted report including the sensor values. Consumers can access this data using a normal web browser, but must periodically refresh the page to see updates. We could easily support the equivalent of an Internet "push" server by including a Java applet on this page that could either connect back to the server using TCP to fetch updates continuously, or connect to some other service such as the Isis MDS (if security is configured to allow this).

Alarms create a publish/subscribe data pattern which is more closely matched to a distribution service such as the Isis MDS. Recall that generic alarms specify lists of applications or users to notify if the alarm condition triggers. A simple way to implement this behavior is to have each such alarm correspond to a publish "subject". Any interested process can simply subscribe to that subject, and any process needing to raise the alarm can do so by publishing a message to it.

THE AGENT PATTERN

PROBLEM

The StormCast system evolves both as we, its developers, add functionality and replace existing components with improved ones, and as the application developers and users "program" the system to ask it to do things. A mechanism is needed for dynamically installing such new functionality, and for removing existing functionality no longer needed. For reasons of performance, the mechanism must support the placement of computation close to the data being processed, and because StormCast is decentralized, the mechanism must be one that can deal with distributed computation.

ALSO KNOWN AS

Object migration, process migration, remote function evaluation, active databases, program installation.

CONTEXT

StormCast is really more of a toolkit than a system, in the sense that the core system exists only to be customized by user-developed applications. A consequence is that new types of data are constantly being added, along with new utility functions to preprocess that data and new applications that operate upon it. Key elements of StormCast include widely distributed servers that archive huge amounts of data, which can easily exceed the capacity of communication links, hence computation must occur close to the data.

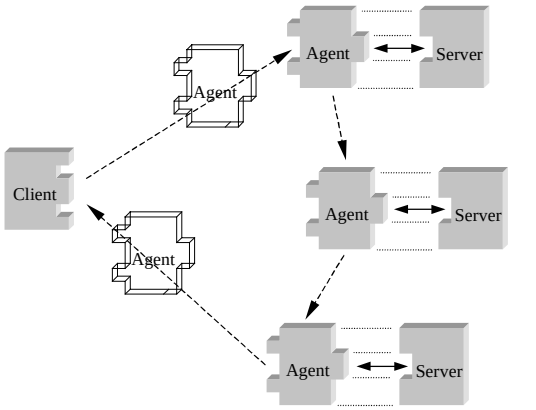

Figure 7: A client spawns an agent, which autonomously visits and computes at a number of servers before returning with an answer for the client.

FORCES

Resource management, scheduling, and overall system security are all important issues when developing applications that match this pattern. By enabling the developer to send computation to a remote site, we are also allowing the developer to potential damage that site or overload it. Tools for limiting computation associated with any given agent and for limiting the amount of data it can accumulate in the form of intermediary results are thus needed. Arguing (strongly) in favor of the use of agents are the tremendous flexibility and improved performance that they bring to the system.

SOLUTION

StormCast uses TACOMA, an unusual agent computing environment tailored to our setting. In contrast to the most widely known of the agent languages, Java, TACOMA is focused on the case in which an agent must be sent to the server. Java applets are pulled from the server to the client, where they execute (perhaps connecting back to the server in the process of doing so). TACOMA brings us considerable flexibility and power, and is designed to support computations that move from place to place "carrying" various forms of data with them. However, resource management and security represent important challenges for us, and TACOMA's solutions for these areas only address some of the issues we face. One consequence is that the use of the agent pattern requires sophistication on the part of the application developer, and great care must be taken to deploy only very robust agents within the current version of StormCast.

EXAMPLES

Agents have been mentioned in connection with several other patterns, including the Assembly Line pattern and the Alarm pattern. A different sort of example was cited in conjunction with weather prediction. In a case where a weather front is approaching some locale, the prediction of the weather for that location may require that computation be dispatched to the server at which the current data gathered for that front is archived, and this process may then iterate as still further servers are visited. Yet another example arose when we worked with a StormCast user who was developing a

commercial application to archive and sell annotated geographic and satellite data (such data is of great value in mineral and oil exploration). In this case, an agent interface permits a potential client to "browse" within the database without retrieving all data items that might possibly be of interest (and incurring a potentially high cost for each item, whether it is genuinely useful or not!). Using agents, the client can design a customized retrieval application that selects and retrieves only data matching some application-specific criteria. The agent language gives much greater flexibility than would any "generic" database interface in this case.

COMMENTS

Performance has been a very serious issue to us in developing TACOMA. Although the basic performance of TACOMA is good, the need to garbage collect forced us to clean up after each agent completed execution, and hence to potentially download the same agent repeatedly if a user requests a single action many times. This can be costly because of TACOMA's security checks and downloading mechanism, which includes (if desired) local code compilation and linking. Recently we have developed code caching schemes that greatly reduce these overheads, and agent programming has become a basic element of the core StormCast system. As just noted, we are allowing users to develop agent applications too, but only under tight scrutiny.

THE CONTROLLER/WORKER PATTERN

PROBLEM

This pattern arises when a task can be performed more efficiently by dividing it into a set of subtasks and distributing these to a set of workers. The pattern is used to speed up computation when an intrinsically parallel operation is being performed within the network, or to do searches distributed over multiple StormCast servers. (We avoid the use of the common term "master/slave" because, for many readers, a master/slave computation has a strong prior connotation).

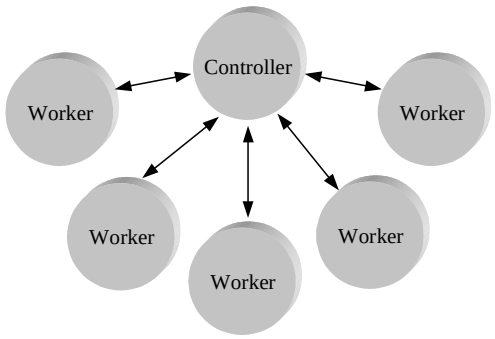

Figure 8: A controller schedules a number of jobs to a cluster of workers.

ALSO KNOWN AS

Master/Slave paradigm

CONTEXT

StormView is a StormCast application used to render 3D projections from weather prediction data. These data are computed at the Norwegian Meteorology Institute (DNMI), and involve a rendering algorithm that is extremely compute-intensive and easily parallelizable.

FORCES

Since the goal of this pattern is to speed up certain computation, the focus must be on efficiency, and especially on the ratio of communication between workers to computation that can be performed at them. We, like others in similar domains, have had the best results where communication is minimized. StormView plays an important role within our system and hence fault-tolerance has emerged as an important attribute of the solution. The failure of the controller or of a worker subprocess should not cause the overall StormView computation to "fail" or "hang."

SOLUTION

Using the Isis Toolkit, mentioned earlier, we implemented a parallelization library over which StormView was implemented. In our solution, reliable group multicast is used by the controller to interct with its workers, and group membership change notifications, which are automatic within Isis, are used to trigger failure handling codes when a participant in the computation fails. (Our solution is tolerant of 1 failure, but not many simultaneous failures). Workers perform rendering subtasks as directed by the controller, which continuously maintains a list of tasks still to be completed. The design is such that a worker can move from one node to another, for example if the load on a node proves unacceptable, or if a job proves to be too computationally costly to handle alone.

EXAMPLES

The StormView system and its use of the Isis Toolkit is described in [Asplin1995]. The parallelization library used by StormView is specialized to that application and has not been made available to other application developers who confront this pattern. However, StormCast is compatible with the use of other parallelization packages, such as PVM or MPI, and with the use of parallel computing libraries or compilers.

THE SCRIPTING PATTERN

PROBLEM

John Ousterhout recently argued that modern computing environments have become extremely complex to program and that much of this complexity is of a tedious, rote nature [Ousterhout 1997]. He suggests that the use of high level scripting languages is more natural, gives productivity improvements, and can be much easier for unsophisticated users or developers. In early years of our effort, StormCast was programmed almost entirely in C and C++, but as time passed, we came to recognize Ousterhout's point. Today, StormCast itself makes use of a variety of programming languages and mechanisms. A high level solution is needed to integrate the resulting components into useful applications.

ALSO KNOWN AS

High-level programming, application prototyping languages, system integration tools, 4th generation programming support.

CONTEXT

StormCast contains a tremendous number of executable subsystems and components, programmed in a number of languages, and sometimes interfaced to genuinely arcane hardware or communication components, such as satellite links. To combine these pieces into useable subsystems, we needed a loosely typed programming language with which to glue them together. We also needed to be able to test the behavior of small, ad-hoc assemblages of components during the development process, and to prototype new components rapidly and without re-implementing large amounts of existing code.

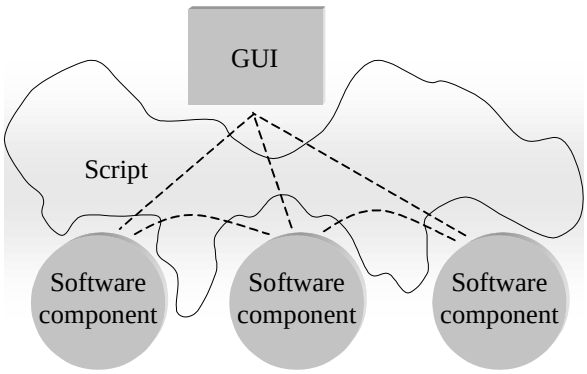

Figure 9: Various software components are glued together using a high level scripting language.

FORCES

The complexity of implementing distributed computing applications directly from low-level components has risen steadily since the early 1980's, and dramatically so in the last two or three years. Although high level languages such as Java are designed with distributed computing in mind, the form of distributed computing that they are intended to support is actually quite remote from the problems we face in StormCast, with its large decentralized data repositories, agents, and irregular communications infrastructure. Even sending a single message can require implementing a substantial infrastructure on a sophisticated modern operating system, such as Windows NT.

SOLUTION

We have used scripting languages as much as possible within StormCast, and encourage application developers to do so as well. When performance is not the primary concern, scripting languages ease the task of adding functionality to existing component sets, and shortens the maintenance process later. Merging components written in different programming languages is easy, which is not necessarily the case in environments like CORBA, which perform rigorous interface type checking. One should, however, take care to select the right language. We use Perl heavily, but find it hard to extend with functionality written in C. In contrast, Tcl/Tk is easy to extend, but our less experienced users find it harder to learn.

EXAMPLES

Most StormCast user interfaces are defined in HTML, and most interactions with server or agent processes are performed using CGI-scripts written in the Perl programming language. The components written in system programming languages such as C and C++ have already been extensively tested, either in prior versions of StormCast or by users in the Internet community. Examples of non-scripted subsystems include TACOMA, mentioned several times above, and PostgreSQL, used as a database tool. Scripting with Tcl/Tk has been especially useful in developing more complex user interface prototypes, but as HTML tools have evolved, they seem to be displacing this requirement.

COMMENTS

Over the course of the StormCast effort, scripting languages have evolved enormously, and the performance of our computing systems has soared. Jointly, these factors have eliminated performance as a major drawback to scripting. Where performance remains an issue, such as for loops or computations that must test data within stored objects, we have been successful in extending Tcl/Tk with code fragments written in other languages, such as C and C++.

[Au: The "side-bar" sections that follow are intended to be typeset as free-floating text boxes, with assoiciated figures in two cases, and need not appear at this specific point in the paper.]

SIDEBAR 1-SATELLITE IMAGES

DESCRIPTION

The StormCast 5.0 satellite services give the users the ability to search for pictures with certain attributes, such as stormy, cloudy or clear weather within a specified geographical region. The specific weather attributes in different regions are stored as a form of meta data that "tags" each satellite image. Preprocessing to create these meta-tags is performed using mobile agents in conjunction with an application based on neural networks. The match with the agent design pattern arises for several reasons. First, satellite data sets are huge, typically 20 Mbytes per image. It is therefore impractical to transfer each potentially interesting satellite data set to the user's computer system for local processing, at least given current network technology. Additionally, there are situations in which the data set is too large to store locally on the client side. Moreover, in the current version of StormCast, some network links are unreliable, causing frequent retransmissions and delayed latency.

As was noted in discussion of the agent pattern, we have worked with satellite image providers who are interested in selling their images. In our approach, the end-user can compute meta-tags for images without transmitting them back to the home system. We ship mobile code to the data set provider, do the computation there, and return the resulting meta-tags. The latter information is usually very small and is easily stored in our storage system. Indeed, the satellite image ultimately provided to the end-user need not be "complete" in the sense that it may even omit some of the data needed to do the meta-data processing.

RELATED PATTERNS

Agent Pattern

SIDEBAR 2- RPC COMBINED WITH PREFETCHING

DESCRIPTION

Some parts of StormCast exhibit a "confusing" resemblance to classical software structures, which can steer the application developer towards an inefficient implementation. For example, StormCast supports several types of remote sensors that only return data when asked to do so. An important case of this concerns remote sensors that reside on trawlers or oil platforms in the Arctic Sea. We have a number of such sensor systems, connected to StormCast's servers by satellite links (Figure 10). These sensors can log more data than the capacity of the link, hence we find it necessary to program the sensor to transmit only weather parameters of current interest. And, obviously, the set of interesting parameters can change depending upon the conditions and current forecast.

Such an application has a resemblance to RPC, and many StormCast application developers have had some experience with RPC solutions. However, when we used an RPC to prototype an application that actually interacted with remote trawlers, we found to our dismay that typical round-trip times could exceed 4 minutes [Farstad 1994]! A much better solution involves preprogramming the remote sensors to periodically transfer weather parameters of interest, with a choice of parameters and frequency controlled by the user. This information is piggybacked on other communication from the trawler to the mainland and cached in an appropriate StormCast server, where it can be accessed by the user or processed in the utility layer. The user's application continues to employ RPC, but only "cache misses" require an interaction with the remote trawler, to reprogram its data transmission profile and parameters. RPC's with cache hits are satisfied in just a few milliseconds.

This structure corresponds to the assembly line pattern (the final RPC off a caching storage layer is so standard that we do not treat it as a special pattern). Specifically, our data collection layer operates by creating a FIFO (first in, first out) cache of time-stamped data received from the monitoring layer. We create stream connections to higher layers when there is some regularity involved. Clients building applications that use this data in higher layers do so by issuing requests to the collection services in which it is archived.

The storage layer services prefetches sensor data periodically to keep availability as high as possible. All other software components interact with the storage layer, not the sensors themselves. If some components (sensors, cameras or software running remotely) generates information that can be reused, this information is stored in the storage system where others may fetch it later based on unique attributes such as time or position. Prefetching data from sensors also has a software engineering related advantage. As no components interact directly with the monitoring layer, we get a uniform access method for storing and retrieving information through the storage layer. This means that services built on top of the storage layer can assume that the storage system always contains up to date information from loggers. If a request is made for data not currently available in the storage system, this is resolved by relaying the request transparently to the monitor layer without the requester knowing it, except through the resulting high latency for the request (if this poses a problem, the application itself can timeout, display a warning, and then retry the request until it succeeds). Solving the problem this way has the advantage is that only services that wish to guarantee low response latencies need to implement strategies for handling (partial) disconnections. We have found it useful to periodically cache data local to utility services that have a pattern of

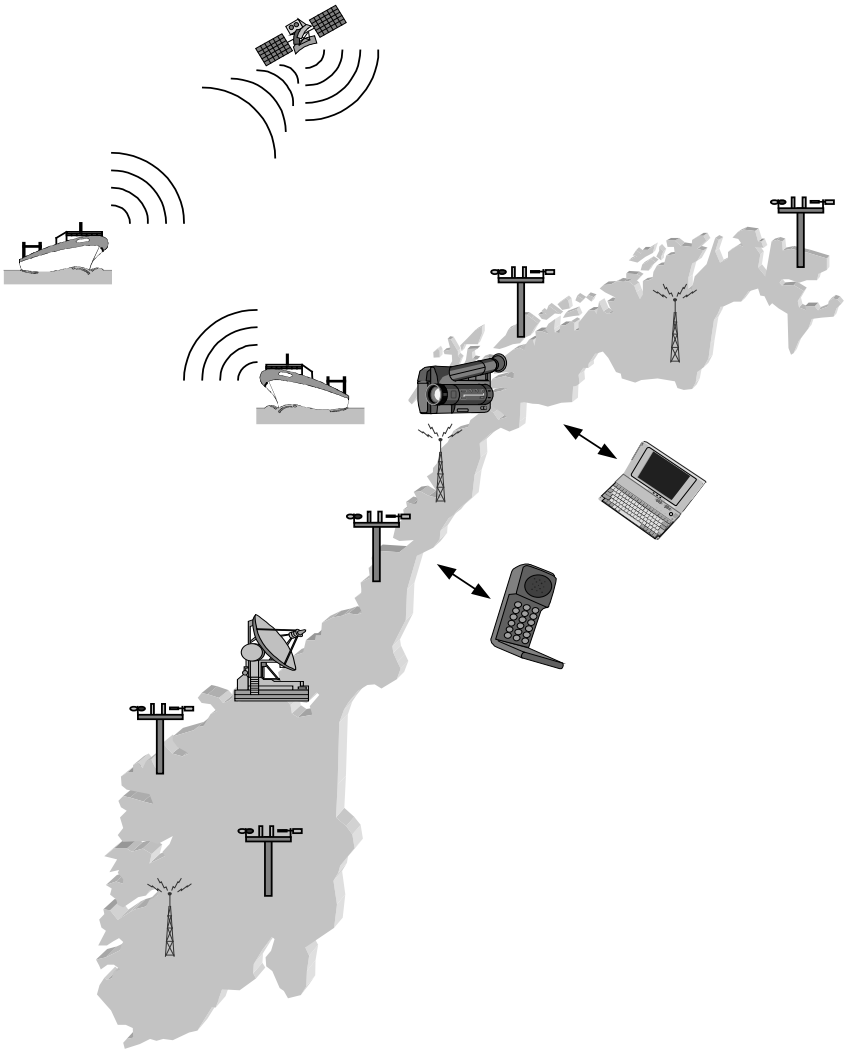

Figure 10: StormCast includes remote sensor systems located on vessels at sea. Latency of interactions may be very high in such cases

requesting such data, as the network distance to the storage system may be long or unstable. To avoid consistency problems, we use read-only caches.

RELATED PATTERNS

Controller/Worker Pattern, Publisher/Consumer Pattern

SIDEBAR 3- STORMVIEW

DESCRIPTION

StormCast gathers tremendous amounts of data related to evolving weather patterns, hence our application developers have invested much effort in finding ways to reduce the data set to an intuitive form that conveys the maximum of information in the most concise format. One such problem in StormCast involves integrating the output of several weather models within a designated time period. Such analysis yields grid data with predicted weather parameters, for instance humidity, wind speed, or wind direction at fixed points in the northern atmosphere. The Norwegian Meteorology Institute (DNMI) typically obtains 230 Mbytes of such grid data each day, and need to summarize them for easy use by human meteorologists. A standard approach has been to produce ISO-charts of individual layers in the atmosphere.

We have addressed this problem by building the StormView volume visualizer, which volume-renders large scalar fields. The input to StormView is a grid object representing the meteorological parameters for the atmosphere, plus a job description (giving the viewing perspective and threshold values). A typical output is seen in Figure 11.

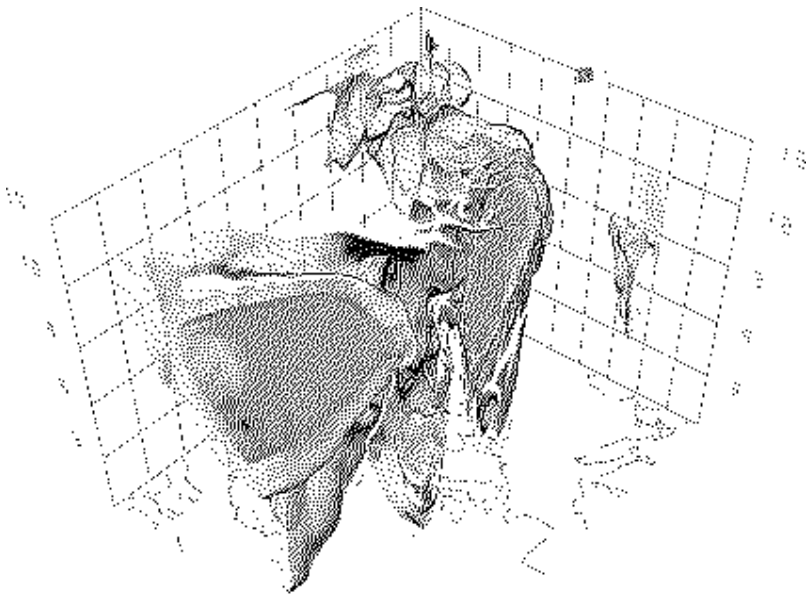

Figure 11: ***StormView example: Windspeed as predicted by a weather model.*** *Remark!* ***For the final paper we can use color versions of this (or any other) image(s).***

Volume rendering is computationally intensive. As such, StormView has been built to run on clusters of workstations. We typically achieve near linear speedup when running on a set of HP 730/9000 workstations, interconnected with a normally loaded 10 Mbits/s Ethernet. The rendering job is split among a set of workers (typically one worker per workstation), and as the rendering progress, the controller gets the partial results from the workers. StormView is fault-tolerant in the sense that if a worker fail, another worker will be given the job instead. The application was built using the Isis Toolkit and follows the controller/worker pattern.

RELATED PATTERNS

Controller/Worker Pattern, Agent Pattern

CONCLUSIONS

In developing the StormCast distributed system, we faced the problem of guiding application developers to the most appropriate application architecture, without exposing them to the detailed internal architecture of StormCast itself. Failure to select the right architecture can result in very inefficient solutions. Our approach to this problem involves using software design patterns as a tool for providing such guidance to the developer. Each pattern represents an architecture StormCast supports effectively, and one to which we are "committed," in the sense that the architecture will continue to be supported in future versions of StormCast.

At present, our work with design patterns is purely pedagogical: we use them to teach the user how best to approach a distributed computing problem. The patterns all correspond to existing StormCast utilities or applications, hence the user who finds a good match with a pattern most often develops their application by copying some existing solution and modifying it to fit application-specific goals. Over time, we believe that this could give rise to a development methodology based on "wizards", similar to the ones used to develop GUI's on modern windowing systems. Even without such software support for carrying out the implementation itself, however, we are convinced that software design patterns offer a way to present StormCast's capabilities to potential users so as to minimize their need to understand internals of the system. By following a pattern, the user maximizes his or her chances of picking an application architecture that will be well matched to StormCast features.

ACKNOWLEDGEMENTS

The insights gained in working with StormCast presented here reflect the contributions and hard work of literally hundreds of colleagues, researchers and students. Although this list is too long to reproduce in the current article, we are profounding grateful to all of them.

REFERENCES

[1] Aarsten, A., Brugalio, D. Menga, G. Designing Concurrent and Distributed Control Systems. *Commun. of the ACM*, Volume 39, Number 10, October 1996.

[2] Birman, K.P. The Process Group Approach to Reliable Distributed Computing. Communications of the ACM, Vol. 36, No. 12, Dec. 1993, pp. 76-84.

[3] Birman, K.P. *Building Secure and Reliable Network Applications*. Prentice Hall / Manning Publishing Company, January 1997. http://www.browsebooks.com/Birman/index.html

[4] Boasson, M.. The Artistry of Software Architecture, IEEE Software, (special edition), Vol. 12, No. 6, November 1995, pp. 13-16.

[5] Farstad, W., Johansen, D., and Hartvigsen, G. Satellite Communication Applied in a Distributed Application, Computer Science Technical Report, 95-18, University of Tromsø, Norway, December 1994. URL: http://www.cs.uit.no/Lokalt/Rapporter/Reports/9418.html.

[6] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., Design Patterns – Elements of Reusable Object-Oriented Software, Addison-Wesley Professional Computing Series, Sixth printing, April 1996, ISBN 0-201-63361-2

[7] Garlan, D., Research Directions in Software Architecture, ACM Computing Surveys, Vol. 27, No. 2, June 1995, pp. 257-261.

[8] Hartvigsen, G., and Johansen, D., Cooperation in a Distributed Artificial Intelligence Environment - the StormCast Application, Engineering Applications of Artificial Intelligence, Vol. 3, No. 3, September 1990, pp. 229-237.

[9] Johansen, D., and Hartvigsen, G., Architectural Issues in the StormCast System, Lecture Notes in Computer Science - 938, (eds. K.P. Birman, F. Mattern, and A. Schiper), (selected papers Dagstuhl Seminar), Springer-Verlag, 1995, pp. 1-16.

[10] Johansen, D., Van Renesse, R., and Schneider, F.B., Operating system support for mobile agents, Proceedings of the 5th. IEEE Workshop on Hot Topics in Operating Systems, Orcas Island, Wa, USA (4th-5th May, 1995), Published by: IEEE Computer Society, NY, USA, May 1995.

[11] Johansen, D., Van Renesse, R., and Schneider, F.B., Supporting Broad Internet Access to TACOMA, Proceedings of the Seventh ACM SIGOPS European Workshop, Connemara, Ireland, (9th-11th September 1996), pp. 55-58.

[12] Ousterhout, J. Scripting: Higher-Level Programming for the 21st Century. Sun Microsystems Laboratories White Paper, Draft #2, May 10, 1997. http://www.sunlabs.com/people/john.ousterhout/scripting.html.

[13] Riecken, D. (guest editor), Intelligent Agents, Commun. of the ACM, Vol. 37, No. 7, July 1994, pp. 19-21.

[14] Scmidth, D., Fayad, M., Johnson, R. Introduction to the Special Issue on Software Patterns. Introduction, *Commun. of the ACM*, Volume 39, Number 10, October 1996.