Convenient Abstractions in StormCast Applications

Dag Johansen[1]

Department of Computer Science
University of Tromsø, Norway
and Cornell University, USA

Gunnar Hartvigsen

Department of Computer Science
University of Tromsø, Norway

Abstract

In this paper we present experience with meteorology applications and appropriate distributed computing abstractions. We focus on the need for co-existence and integration of multiple paradigms in large scale distributed applications, rather than enforcing a favourite paradigm whenever possible.

1. Introduction

A main objective of the StormCast project is to identify appropriate abstractions for distributed application programmers. Specific application needs and real computing problems are solved through an interdisciplinary approach involving computer scientists, electrical engineers, meteorologists, air pollution specialists, as well as industry. So far, more than 100 students and faculty have participated over the last 8 years. A central problem has been the development of a general software architecture [Joha 93a], an architecture that has matured over multiple implementations [Joha 89][Hart 90][Joha 93b]. Of particular interest is the need to fully automate the entire meteorology process, including weather data monitoring, collection and storing, weather prediction, and visualization. These solutions must be scalable, fault-tolerant and meet demanding performance requirements.

We have applied well known distributed computing abstractions as client-server computing and group communication to computing problems in this domain. This paper briefly presents some of the abstractions we found useful. It is organized as follows. First, we briefly characterize the meteorology domain in a distributed computing perspective. We then discuss the most useful abstractions for distributed computing. Finally, we conclude by focusing on the need for combining different distributed computing abstractions in the same application.

2. Characteristics and Challenges in the Application Domain

An applied approach requires that we understand the problems and limitations in the domain. Weather prediction as done by the Norwegian Meteorological Institute works as follows. Manually operated weather stations deployed throughout Norway (in the order of hundreds) provide raw weather data such as temperature, pressure, wind speed and wind direction. This data is transmitted to three centralized sites through phone and fax 3 to 8 times per 24 hours. This is supplemented by automated weather stations, weather buoys and weather balloons. Part of this raw data is input to two weather prediction models computed on a CRAY Y-MP. This raw and processed data, combined with satellite images, are used by meteorologists to predict the weather for the next day or, as often done, five days ahead in time. Unfortunately, little tool integration exists.

The current focus in the StormCast project is to prototype a meteorological workbench. This is a large scale multi-media application integrating and automating the whole process from data

1. This work was supported in part by the Norwegian Science Foundation (No. 100413/410).

monitoring to visualization. Current snapshots of raw data from remote monitoring devices, historic data and statistics can be obtained. A distributed expert system predicts severe weather conditions, and a pattern matching utility predicts weather by combining current and historic weather data. Satellite images can be integrated as well as video from remote, unmanned locations. Finally, 3D visualization techniques are used to present output from weather models run on remote mainframes. We are currently adding virtual reality techniques to this 3D space.

Such an application operates in a very heterogeneous environment with computers ranging from small dedicated loggers out in the field, to mainframes and clusters of workstations. Devices such as sensors and video cameras add to this diversity. Networks include high latency satellite networks running X.25 at 600 baud to high-performance local area networks. Data sets are in the order of 50 bytes for a site report, to 200 Mbytes of output from the CRAY per 24 hours. Programming languages include C, C++, Common Lisp, Visual Basic, Forth and assembly. Operating system software includes UNIX, ISIS [Birm 93], DOS and custom built systems. This diversity is illustrated in Figure 1.

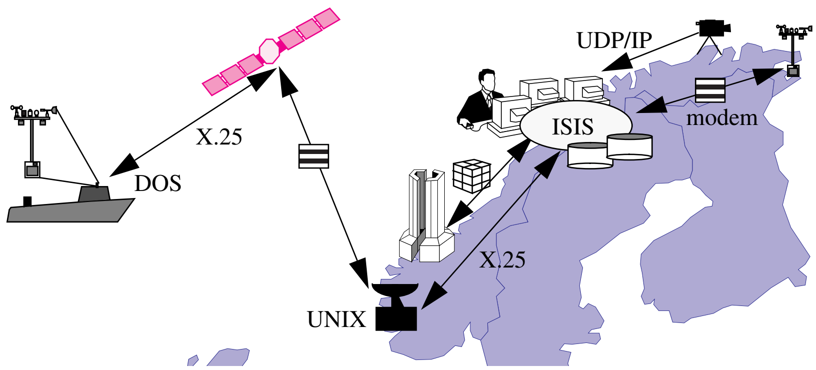

FIGURE 1. The StormCast Meteorological Workbench.

The quality of service needed here is typical for a soft real-time monitoring application. Increased fault-tolerance is a key concern, especially since we use high latency networks with low availability. The applications must also be designed to scale well. Finally, performance is, as usual, a design impediment.

3. Experience with Some Abstractions

Building scalable, fault-tolerant, high performance applications in this environment, requires design and implementation trade-offs. Some abstractions have shown to be more convenient than others. We have made the following abstraction observations in our applied work.

First, we must select the *appropriate tool* for each problem, rather than enforcing one tool to solve every problem. We do not necessarily need the same communication protocol or operating system throughout an application. For instance, we use the ISIS Toolkit in a network of HP 735 and 755 workstations. This provides better fault-tolerance inside this cluster, but does not impose extra complexity outside. External communication is through other protocols such as X.25, UDP/IP, or our own RPC [Birr 84] and multi-RPC protocols [Saty 90]. Similarily, a pure UNIX platform might be convenient on a high-end workstation, but too heavy-weight to be run on a logger out in the field. We have built a small multi-tasking operating system for such a logger embedding the logger server in one functional unit. The functionality of this *microserver* is to poll connected sensors and handle external communication. This is all functionality that is needed out there.

This heterogeneity also implies that *multiple interaction paradigms* can be successfully integrated in the same application. For instance, we find it useful to combine a client-server approach with a more stream-oriented approach. Client-server interactions are preferred for communication between modules located in the same local area network (LAN). A service can be requested only when needed, something that scales better than a stream-oriented approach. A StormCast application, however, crosses LAN boundaries. A LAN based RPC is in the order of 3 milliseconds. A modem based RPC is in the order of 36 seconds. We have added a stream-oriented approach where data is periodically transmitted into this LAN environment. This prefetched data can be validated (its time stamp) before any external request is issued. This is a potential performance advantage, but is also a potential scaling problem. The problem is to determine where the raw data should be sent and how often. Data in this domain is valid (as current data) as long as 20 minutes, so the transmission frequency does not have to be higher than this. If this cached data is invalidated, a client-server interaction fetches the current data.

Prefetching data introduced the need for *caches*. Data in this domain is *immutable* by nature. This simplifies the complexity caching naturally imposes on an application. We added caching structures throughout the applications, basically at physical domain crossings with low latency. For instance, a RPC to a logger on a boat in the Arctic Sea has a round-trip delay in the order of 4 minutes through the Inmarsat.c satellite system. This comes in addition to connectivity problems in the very far north. A protocol between the mainland and the boat has been developed to prefetch time stamped data for mainland located caches. This meteorology data can also be piggybacked on other communication. A request with a cache hit is now in the order of 3 milliseconds to 2,9 seconds. The first uses a cache at the same local area network; the last must access a remote cache at the satellite earth station through X.25 connections. Whether a cache is used or not is transparent for the user. Figure 2. illustrates mean latencies measured for *get_data(time, location; weather_params)*, one of the most frequently used interface abstractions. Notice that the same abstraction is used to obtain data from remote loggers (or local caches), as well as historic data stored on multiple, local disks.

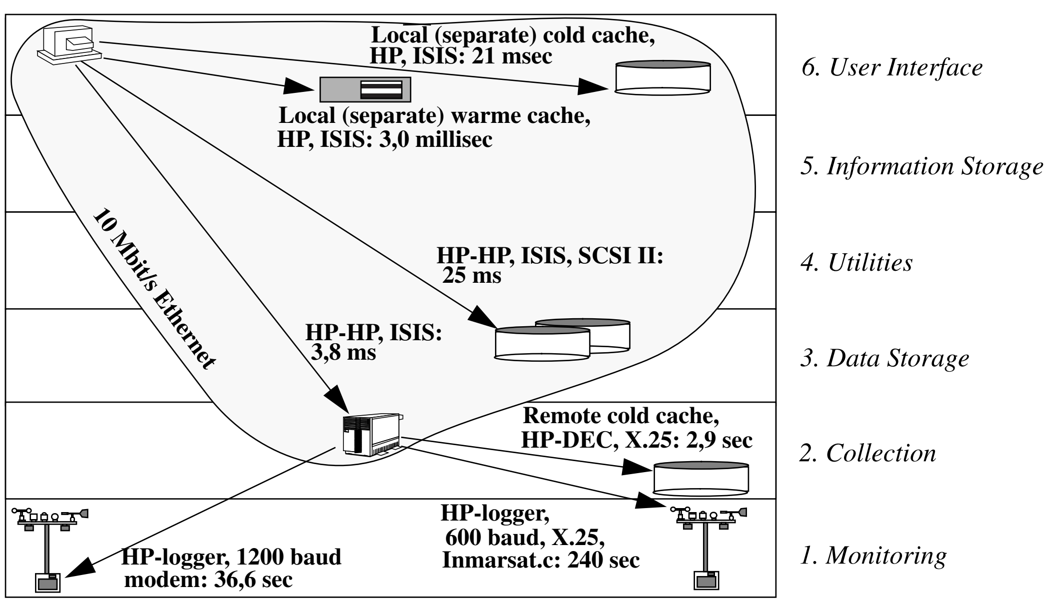

FIGURE 2. get_data(50 byte) round-trip latencies.

Caching is useful at *both sides* of a low available connection. Hence, we also cache data in the loggers. This is convenient for data recovery after a disconnection. A server providing permanent storage of periodic weather data, both for research and later prediction purposes, might need data in this remote cache upon reconnection.

Proxy structures [Shap 86] provide basic functionality of a remote service in case of disconnections with the rest of the application. We find this abstraction useful, especially using it as a service provider [Joha 93b] in the front-end of an application.

Data *filtering* is convenient in StormCast applications to reduce the amount of data. The problem is not to obtain enough data. The problem is to reduce this amount to useful information within a certain time. Gigabytes can be reduced to a few bytes. This filtering should be done as close to the data source as possible. This assumes that all users agree on the quality and functionality of the remote filtering process. A logger is therefore capable of running filters as alarm functions. The functionality is to check for dramatic weather changes and notify this to external receivers. This off-loading of computing from higher layers is a scaling factor since these layers can reduce the amount and frequency of data otherwise transmitted.

Disconnection problems are the main source of failures in StormCast applications. In addition to caching, we approach this problem through *emergency* protocols. These are protocols only used in exceptional situations. For instance, a logger is likely to become disconnected through the regular satellite communication path. This logger can have detected some severe change that has to be reported immediately. The logger can now use packet radio transmission protocols and send its data to neighbouring loggers. These will act as relay stations, either forwarding this data to the next logger, or attempting to send it over their satellite link. This adds complexity and is used with caution.

Replication techniques have also been useful for increased fault-tolerance in StormCast. Reliability is provided as close to the data source as possible. Once correctly captured, immutable data will remain correct. This also reduces the amount of data needed to transmit (or retransmit). Some of these sensors exhibit Byzantine behaviour, especially during freezing and humid conditions. It is, for instance, hard to determine if a remote sensor is covered by ice and snow for a short period of time. Such a coverage influences the output from sensors with mechanical parts like the wind speed and wind direction sensors. Sensors like the rainfall sensor and conductivity sensor are also problematic. Sensor replication, with each replica mounted differently, and voting mechanisms is an approach to achieve increased reliability. To our surprise, end users did not always want such replication transparency. They wanted to decide themselves which value to actually use. At the same time, knowing about such problems was valuable knowledge in itself. However, utilities using this data need a single value, not a set of values.

The ISIS Toolkit provides an environment for server replicas. Failure-prone parts of the StormCast applications are executed in this LAN environment. We find the *group* abstraction very appealing for this. A service is implemented by a group of servers based on either active or passive replication techniques. Preferring one over the other is often determined by the need for requests outside the LAN. If a service must issue a nested request crossing such boundaries, we favour *passive replication*. Only the primary replica is active sending one request, with one of the backups activated upon primary failure. If no such external redundancy is involved and there is a need for redundancy, as for a data storage operation, we use *active replication.*

We increase performance through *subdivided* computations. This is a convenient abstraction when treating a network of workstations as a multi-computer. A group of workers is managed by a controller. The number of group members might vary throughout the time a service is provided. This is managed by ISIS. The controller splits up a computation and sends it to the workers. The workers send their results back to the controller for assembly of the overall result. A new request for more work can now be piggybacked on this reply. We achieve almost linear speedup when running a compute intensive ray casting visualization server on, for instance, 25 workstations in parallel [Aspe 94]. A key factor here is to avoid *synchronization points* between the individual workers and to use *server-initiated* scheduling. Consequently, the individual worker requests work from the controller only when it is ready.

Experience also indicates that network performance does not have to be a bottleneck. A parallel implementation of our visualizer consists of **n** workers and a controller. A job split up in **m** tasks (pixel sets) normally requires **(m+n)** messages in addition to an initial broadcast. ISIS overhead is in the order of a millisecond per message. Hence, it is negligible to the overall linear performance achieved here.

Atomic delivery is not always necessary in this domain. We have implemented multi-RPC protocols where both a request and a reply can get lost. This is something likely to happen in a satellite environment in the very far north. The validity of data in this domain is often inverse proportional to real world time. Consequently, it can be more important that some group members receive a message rather than enforcing everybody to wait until the entire group is available.

The very loosely coupled environment for StormCast applications does not favour distributed shared memory abstractions. Nevertheless, we find a *partially shared memory* abstraction convenient. A group of loosely coupled, yet cooperating expert systems, interacts through such an abstraction. All expert system modules are equipped with a blackboard module [Nii 86], each expert system module responsible for predicting weather in its geographical area. Whenever weather is predicted by an expert system E_1, it broadcasts this information to a group of neighbouring expert systems. A receiving expert system E_2 uses its local blackboard to store this. E_2 can use this information later when its local prediction is done. A client-server based cooperation approach here could cause a deadlock situation. For instance, an expert system E_1 issues a blocking request to E_2. The problem now occurs when E_2 again must issue a request to E_1 to handle this request.

Finally, we would like to comment more generally about how to structure applications that are inherently distributed. It is important that application developers decompose such an application from the start. Beginning with a centralized structure often ends up in ad hoc solutions not matching the application needs. If necessary, two modules can later on be merged into a single computational unit for performance reason. This has shown to be trivial compared to the opposite approach. A *general architecture* for this type of application aids in the structuring process [Joha 93a]. A *general design* for this type of applications is also convenient [Joha 93b]. This has resulted in a higher layer StormCast API with reusable library routines.

The rule is to decompose StormCast applications into three general components. One part of an application deals with I/O and nothing more. User interfaces of StormCast are cpu-intensive enough to each utilize a single workstation in itself. Similarily, we consider a monitoring module a specialized I/O device. Our second type of modules are compute-intensive services. These can be run externally to the interface part of an application. A third module type is related to data storage operations.

5. Concluding Remarks

We have developed a new generation of scalable, fault-tolerant application prototypes in real domains. A main lesson is to combine different paradigms for distributed computing. The same application can benefit from both group communication protocols as well as more traditional RPC protocols. The same application can utilize both shared memory abstractions as well as message-oriented approaches. We strike a balance between client-server interactions, a stream-oriented approach to fill caches, and computing close to the data source.

To take full advantage of a distributed platform, application programmers must still be responsible for selecting the appropriate abstractions and where to locate data and computing. Performance of inter-process communication does not have to be a bottleneck in this environment. Effective decomposing and off-loading of centralized tasks can make up for performance penalties paid. This approach eases scaling as well as fault-tolerance.

To match application needs in a heterogeneous environment, domain specific, high-level APIs should be provided. Application programmers can then focus on mastering the semantic problems of such large scale applications, a challenge in its own right.

Acknowledgements

The StormCast project is a team effort. Recent contributions by Jo Asplin, Vahid Farsi, Werner Farstad, Bård Fjukstad, Bente Høgtun, and Pål Knudsen have been important. The work presented here also benefited from discussions with Kenneth P. Birman, Fred B. Schneider, and Robbert van Renesse. We thank Robbert, Marc Shapiro, and anonymous referees for their helpful comments on the paper.

References

[Aspe 94] Aspelin, J. and Johansen D. Load Sharing in a Distributed Parallel Volume Renderer. *in preparation*.

[Birr 84] Birrel, A.D and Nelson, B.J. Implementing Remote Procedure Calls. *ACM Trans. Computer Systems*, 2, 1 (Feb. 1984), pp. 39-59.

[Birm 93] Birman, K.P. The Process Group Approach to Reliable Distributed Computing. *Commun. ACM*, 36, 12 (Dec. 1993), pp. 36-53.

[Hart 90] Hartvigsen, G. and Johansen D. Cooperation in a Distributed Artificial Intelligence Environment - The StormCast Approach. *Eng. Appli. of AI Journal*, 3 (Sep. 1990), pp. 229-238.

[Joha 89] Johansen, D. Weather Forecasting - Distributed in Nature. *Proceedings of IFIP TC6/TC8 Open Symp. on Net. Info.Proc. Sys.*, (May 1988), North-Holland (1989), pp. 197-203.

[Joha 93a] Johansen, D. StormCast: Yet Another Exercise in Distributed Computing. In *Distributed Open Systems*, (eds. Brazier, F., and Johansen, D.), IEEE Computer Society Press, NY, USA (Oct. 1993), pp. 152-174.

[Joha 93b] Johansen, D. A Distributed Approach to the Design of Applications. In *Proc. of 5th Int. Conf. on Comp. and Inf.*, Sudbury, Canada, (May 1993) Publ: IEEE Computer Society Press, NY, USA (Nov. 1993), pp. 195-201.

[Nii 86] Nii, H.P. Blackboard Systems: The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures. *AI Mag*., 7, 2 (Summer 1986), pp. 38-53.

[Saty 90] Satyanarayanan, M. and Siegel, E.H. Parallel Communication in a Large Distributed Environment. *IEEE Trans. Computers*, C-34, 12 (Dec. 1990), pp. 328-348.

[Shap 86] Shapiro, M. Structure and Encapsulation in Distributed Systems: The Proxy Principle. *Proceedings of ICDCS,* IEEE Computer Society Press, (May 1986), pp. 198-204.