ADHOC design principles.

ADHOC (Adaptive Distributed Herd of Object Caches), is a distributed object repository. It provides applications with a distributed storage manager that virtualizes Processing Elements primary or secondary memories into a unique common distributed data repository. However, it is not just another Distributed Shared Memory (DSM), it rather implements a more basic facility. The underlying idea of ADHOC design is to provide the application (and programming environment) designer with a toolkit to solve data storage problems within Cluster Of Workstation (COW) and more generally in the Grid framework. In particular, it provides the programmer with building blocks to set up client-server and service-oriented infrastructures which can cope with Grid difficult issues aforementioned.

The semi-finished nature of ADHOC ensures high adaptability and extendibility to different scenarios, and rapid development of highly efficient storage and buffering solutions meeting scientific and industrial needs. The ADHOC underlying design principle consists in clearly decoupling the management of computation and storage in distributed applications. The development of a parallel/distributed application is often legitimated by the need of processing large bunches of data. Therefore data storages are required to be fast, dynamically scalable and enough reliable to survive to some hardware/software failures.

Decoupling computation and data storage helps in providing a broad class of parallel applications with these features while achieving very good performances.

Clients may access data stored into ADHOC servers using a simple proxy library. Proxies may act as simple adaptors. Both clients and servers may be dynamically attached and detached during the program run.

A set of ADHOCs implements an external storage facility, i.e. a distributed repository for arbitrary length, contiguous segments of data (namely objects). An object cannot be spread across different ADHOCs, it can be rather replicated on them. Objects can be grouped in ordered collections (of objects), which can be spread across different ADHOCs. Both objects and their collections are identified by keys. In particular, the key of a collection specify which spread-group and replica-group the collection belongs to. Adjacent objects of a collection are allocated and stored in a round robin way along a list of replica-groups. Each object is stored in each server appearing in the replica-group. ADHOC API enables to get/put/remove/execute an object, and to create/destroy a key for a collection of objects. ADHOC does not provide collective operations to manage collections of objects (except key creation and destruction), these collective operations can be implemented within the client proxy.

Each ADHOC manages an object storage and a write-back cache used to store server home objects and remote home objects respectively. ADHOC execute method enables the remote execution of a method, provided the key refers to a chunk of code instead of plain data (i.e. an actual object which is executable on the target platform). This operation is meant as mechanism to extend server core functionalities for specific needs. As an example, lock/unlock, object consistency management, and atomic sequences of operations (e.g. get_and_remove) on objects have been introduced in ADHOC this way.

ADHOCs may work as relays for others. This allows to set up a distributed data server across networks with different private address ranges, that is the usual configuration of clusters belonging to a Grid.

For get/put objects, each connected graph of ADHOCs is functionally equivalent to a complete graph. However, currently only directly connected ADHOCs may belong to the same spread- or replica-group (collection cannot be spread through relays). Moreover, since ADHOCs may be dynamically attached, different subgraphs are not supposed to be started all together, as it would happens in the case they are
executed through different job schedulers on top of different clusters.

**Some notes on ADHOC implementations.**

An ADHOC is implemented as a C++ single threaded process; it relies on non-blocking I/O to manage concurrent TCP connections. The ADHOC core consists of an executor of a dynamic set of finite state machines, namely Services, which reacts to socket-related events raised by O.S. kernel (i.e. connections become writable/readable, new connection arrivals, connection closures, event timeout, etc.). In the case one Service must wait on an event, it consolidates its state and yields the control to another one. The ADHOC core never blocks on I/O network operations: neither on read()/write() system calls nor on ADHOC protocol primitives like remote memory accesses.

The event triggering layer works with the Poller interface (see [www.kegel.com/dkftpbench/](http://www.kegel.com/dkftpbench/)) and the libevent library (see [www.monkey.org/~provos/libevent/](http://www.monkey.org/~provos/libevent/)). With the Poller interface it may be configured to use several POSIX connections multiplexing mechanisms, such as: select, poll, and Real-Time signals. An ADHOC server can efficiently serve many clients on a single port, each of them supporting thousand of concurrent connections. ADHOC is a GPL software it has been tested on several Linux and BSD platforms; it supports heterogeneous clusters.

**Installation.**

For installation instructions, please take a look at HOWTO_eg.txt file in the ADHOC/doc directory.

For any tips improvements or bug, please contact
Massimo Torquati torquati@di.unipi.it or
Marco Aldinucci aldinuc@di.unipi.it.
Thanks in advance.