APPLICATIONS OF TRANSPARENT PROXY SERVERS IN CONTROL SYSTEMS*

B. Frak, T. D'Ottavio, M. Harvey, J. Jamilkowski, J. Morris, Brookhaven National Laboratory, Upton, U.S.A.

Abstract

Proxy servers (Proxies) have been a staple of the World Wide Web infrastructure since its humble beginning. They provide a number of valuable functional services like access control, caching or logging. Historically, control systems have had little need for full-fledged proxied systems, as direct, unimpeded resource access is almost always preferable. This still holds true today, however unbound direct asset access can lead to performance issues, especially on older, underpowered systems. This paper describes an implementation of a fully transparent proxy server used to moderate asynchronous data flow between selected front-end computers (FECs) and their clients as well as infrastructure changes required to accommodate this new platform. Finally it ventures into the future by examining additional untapped benefits of proxy control systems like runtime read-write modifications.

INTRODUCTION

In a perfect world proxies are not only unnecessary, but are usually frowned upon, as they introduce additional complexity to already complicated systems. The majority of the proxy-based systems require some level of additional configuration – usually in the top most client layer. Additionally, in ideal conditions performance issues are non-existent, so any latency or throughput benefits provided by caching proxies are also rendered irrelevant. However we do not live in a perfect world and often proxies are the only cost effective way to accommodate growing and/or aging accelerator infrastructure.

The idea to implement a truly transparent proxy-server framework in the Collider Accelerator Department has been kicked around for quite a while. The concept would usually come up while trying to tackle performance issues of one or more struggling front-end computers, which had been over-utilized and thus were unable to handle requested loads. Prior to the development of the transparent proxy system, the solution to these problems usually involved utilizing custom middleware servers, which throttled back the backend side utilization by putting themselves between the clients and the front-end computers. This is still an acceptable solution, however it's not as flexible as its transparent counterpart, as it does require additional, persistent modifications in the client layer. For some applications, these changes can be trivial, for others they require a more substantial time investment. In neither case it is automatic or transparent.

*Work supported by Brookhaven Science Associates, LLC under contract no.
DE-AC02-98CH10886 with the U.S.

Reflective Server (RS) has been designed to circumvent this issue. It features all the benefits of a classic middleware proxy without introducing additional configuration entropy to the system.

IMPLEMENTATION

Reflective Server is built on top of the existing Java Accelerator Device Object (ADO [1]) framework. Its core contains one or more self-configuring, faux device objects, which inherit all the external, system visible features of a true, reflected instance. In essence, this process creates one twin image of each real ADO, which resides on another host (usually a FEC). The similarities are only skin-deep, as the faux instance is just a thin shell masquerading as the real device object. It knows nothing about the business logic of the real instance, however it does know how to forward requests and data to and from its real counterpart. It, similarly to its twin, resides in a container that handles all the client-server communications. The standardized ADO communication RPC [2] vocabulary is also identical, which means that from a client perspective, real and faux systems are indistinguishable. Figure 1 shows both direct and indirect client access pattern. The clients are completely identical, and neither one can tell whether or not they are accessing the data via a proxy or directly.

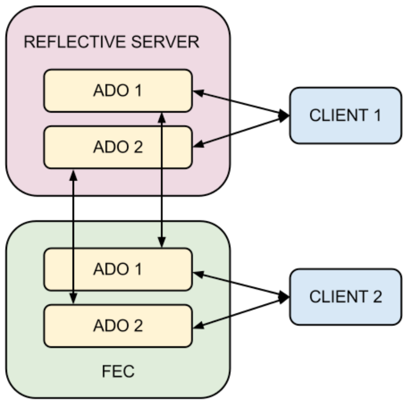

Figure 1: Direct and indirect access pattern.

Transparency

The above-mentioned transparency is achieved by modifying the name server records on as needed basis. The Reflective Server when bootstrapping device objects modifies their entries in the master Controls Name Server (CNS) to reflect their new "location". This record includes a host name as well as RPC program and version numbers required for all client server communication. These records remain unchanged for the duration of proxy

ISBN 978-3-95450-139-7

server lifetime and each Reflective Server instance has a responsibility to restore original entries upon shutdown. Clients always obtain the location of device objects from the name server, which means modifying this central repository is the only way to transparently inject proxy instances to the live system. Figure 2 shows the actors participating in name lookup process. Relational database is used to prepopulate records in CNS. Clients retrieve desired record based on the supplied name. Once decoded the record points to a device object's network location. Reflective Server(s) can at any time modify the name server contents, thus rerouting clients to its location.

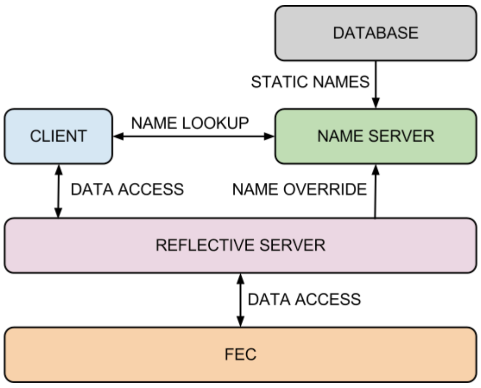

Figure 2: ADO name lookup.

Runtime Configuration

From a Reflective Server perspective Accelerator Device Object is a single indivisible entity. As previously mentioned an ADO server is a collection of these device objects. Proxy servers by virtue of being an ADO container adhere to this principle, which means that can also be configured in variety of ways depending on a problem they are trying to solve. Two of the most common configurations are:

- One Reflective Server per FEC – this is by far the most common scenario. Each server instances maps and reflects all (usually > 100) ADO instances on a selected FEC. This scenario is used when trying to offload a busy front-end computer.
- One Reflective Server for multiple scattered ADO instances – in this scenario one server maps and reflects selected ADOs from one or more FECs. This set-up is used to target a specific, usually very popular, application, whose instance count is causing unusually high strain on accessed FECs.

Other configurations ranging from one ADO per server to all ADOs in the Collider Accelerator Department per single RS instance are also possible - though highly improbable.

Synchronous Access

All synchronous requests for dynamic data go through the proxy servers to the underlying FEC system unimpeded. Reflective Server routes all synchronous RPC requests originating from the client-facing layer to the appropriate ReflectiveAdo instance. This instance contains exactly one RemoteAdo object responsible for the FEC-facing communication. It re-encodes the parameter set and dispatches the request to the appropriate front-end instance. Results are bubbled up to the calling client.

Some operations, such as synchronous reads, can benefit from the built in proxy caching, however this feature is disabled by default. Immutable data is always cached in the proxy.

Asynchronous Access

This is the key area where Reflective Server framework proves to be the most valuable. By positioning itself in front of backend infrastructure, it essentially removes all subscribe-publish related scaling issues. This mechanism relies on proxy instances becoming exclusive clients to FEC server instances, and thus taking the burden of handling all, client issued, asynchronous requests onto itself. Figure 3 shows the default, direct access pattern. In this case the FEC is responsible for all client connections. Figure 4 shows the equivalent, indirect access pattern. In this case the FEC is only dealing with one client, and that is one Reflective Server instance, which in turn manages all client issued asynchronous requests.

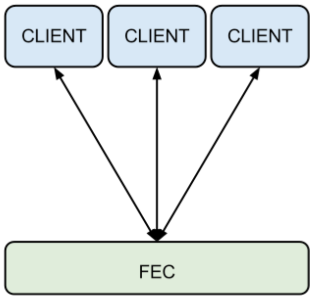

Figure 3: Direct asynchronous connection request model.

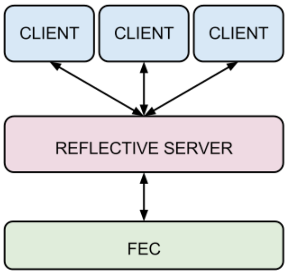

Figure 4: Proxy routed asynchronous connection request model.

Dealing with the Unexpected

Front-end computer reboots, timeouts, backend communication failures, and client dropouts – these are just a few examples of exceptional system states, that have to be handled by the Reflective Server instances. The transparent nature of the proxy servers has to be enforced during these conditions. Clients routed through RS instances must be notified about any failures in the same exact way as if they were connected directly to the FECs.

Front-end communication loss due to a machine reboot is by far the most common exceptional state encountered during normal accelerator operation. Clients receiving data asynchronously have a number of built in mechanisms to identify, mark and eventually attempt to reconnect to a failed host. They rely on both ADO protocol as well as built-in RPC constructs like portmapper to accurately judge FEC states. This behavior has been meticulously tested and it works exceptionally well in direct server access scenarios. However it does introduce additional level of complexity to the proxy servers, as they have to correctly simulate all possible client-observable states. A full-fledged FEC failure due to a reset would trigger the following sequence of events in a previously connected Reflective Server:

- Terminate client facing RPC transport responsible for the affected front-end machine. We want the clients to immediately notice that the FEC is no longer available. The only way to achieve this without actually killing the server process is to shut down part of the front-facing infrastructure.
- Enable more aggressive heartbeat mechanism to speed up recovery process.
- Terminate asynchronous connections to cleanup publish-subscribe bookkeeping structures. Clients automatically reissue requests for their entries when the server comes back online.
- When the FEC is back in service reestablish client-facing RPC transport and wait for requests.

Other exceptional states are also handled accordingly to well known and accepted rule-set.

EXTENSIONS

Reflective Server implementation is based on a new Java ADO platform. This close relationship exposes additional features native to this platform to ADO designers and developers. The most important one being AOP [3] modifier chains, which allow for device object extensions.

Extension is an advice, which cuts across all sets and gets (input and outputs) for all Reflective Server contained device objects. This advice is supplied to the RS runtime as a class file, which gets weaved with the existing set of advices already attached to the ADO set and get methods. The most basic extension point overrides two methods from the base aspect – one for input and one for output modification. The former has full control over the data sent to the slave ADO, while the latter controls the data shipped back to the clients. This tight pairing can be utilized by device object developers in a variety of ways during all stages of a development cycle as well as in deployed systems.

Extensions point flexibility and ease of deployment makes them excellent candidates for analyzing application behavior and mocking up test cases for various system components. ADO developers can effortlessly create artificial ADO states as seen by the application by modifying the data output to the clients. Note that we are not modifying real ADO values - those remain unaltered. However clients accessing them through a Reflective Server with a modifying extension point are oblivious to that fact. To them modified values are real. Figure 5 shows an example where “floatM” parameter is forced to a constant value for testing purposes.

```
@Override
public Float modifyOutput(String device, String parameter,
        String property, byte ppmUser, Float value) {
    if("simple.test".equals(device) && "floatM".equals(parameter))
        return 0.0f;
    return value;
}
```

Figure 5: Output modifying extension point.

Input modifying extensions are equally useful. Figure 6 shows two simple, yet very powerful examples. The first one changes the client interface to all current settings on all reflected ADOs from amps to milliamps. Again we are not changing the underlying device objects – they will still receive values in amperes. The second example removes all write access to the “simple.test” device.

```
@Override
public Double modifyInput(String device, String parameter,
        String property, byte ppmUser, Double value) {
    if(parameter.startsWith("current")) {
        return value * 1000.0;
    }
    return value;
}

@Override
public Object modifyInput(String device, String parameter,
        String property, byte ppmUser, Object value) {
    if("simple.test".equals(device))
        throw new AdoIfModuleException(AdoIfModuleException.ADOIF_WRONG_SET);
    return value;
}
```

Figure 6: Input modifying extension points.

APPLICATION

Several candidates were identified where FEC loading was inhibiting the overall system performance. Adding the Reflective Server layer has improved the FEC utilization in all cases. Lower throughput cases performed seamlessly however higher throughput did encounter intermittent loss of communication.

An example of a low throughput is the Beam Inhibit Reflective Server instance, which collects a small sample of data from four different FECs. This system is heavily used by the Operations, and the proxy server, which has been deployed for the last two runs encountered no issues.

The Low Level RF FECs are much higher throughput with each FEC passing ~5GB of data through this layer per hour. These front-ends were also placed behind a layer of multiple Reflective Server instances in a pseudo-operational model. Proxy servers at these data volumes still require some handholding, as they are not as stable as their low throughput counterparts. Even with frequent interventions, which require either a server restart or FEC reboot, the benefits still outweigh the drawbacks.

FUTURE DEVELOPMENT

At the top of the agenda for future Reflective Server development is improving reliability. In order to make this a truly operational system it will need to become more robust, both in performance and in limiting any impacts upon clients of the RS when problems do arise. Once this improvement is in place, we foresee many more systems making use of this functionality.

REFERENCES

[1] L.T. Hoff and J.F.Skelly, Accelerator Devices at Persistent Software Objects, Nucl. Instr. and Meth. in Phys. Res. A 352 (1994),

[2] SUN-RPC RFC, http://www.ietf.org/rfc/rfc1057.txt

[3] G. Kiczales, J. Lamping, C. Lopes, J. Hugunin, E. Hilsdale, C. Boyapati "Aspect-oriented Programming", (1999).

ISBN 978-3-95450-139-7