THE TINE CONTROL SYSTEM PROTOCOL: HOW TO ACHIEVE HIGH SCALABILITY AND PERFORMANCE

P. Duval and S. Herb, DESY/Hamburg

Abstract

Over the years the TINE control system [1] has implemented numerous strategies for achieving high efficiency data transport within a distributed control system. This was essential for controlling a large machine such as HERA [2]. Our recent experience with controls for the PETRA3 and FLASH accelerator complexes at DESY has revealed new scalability issues. The principal problem has been in limiting the communications load on the front end servers and network in the presence of increasing numbers of client applications, many of which are written by 'part-time' developers who prefer simple API calls, or use development platforms which support only such calls. A single such application, polling hundreds of devices, may generate ~1000 calls per second to a single server. This load on the server can be reduced if, for example, the intermediate software layers can consolidate such calls into array transfers. TINE now offers various 'second-order' protocol features which go a long way toward not just allowing but 'enforcing' efficient data transfer. We shall describe some of these features in this article.

INTRODUCTION

In this report we concentrate on how the control system protocol can be a limiting factor in scalability regarding large distributed systems. To this end it is necessary to review some popular communication strategies along with application programmer interface (API) paradigms.

DISTRIBUTED DATA FLOW

0th Order: Transaction-based Client-Server

The earliest versions of most popular control system protocols made exclusive use of transaction based client-server polling. This data-flow pattern has the inherent advantage of a 'keep it simple' strategy, but can quickly run into scalability issues. These often manifest themselves as server-load problems rather than network-load problems, although both issues are important.

We take the average *load* (per second) on a server due to polling clients to be roughly given by

$$L_S \sim N_c \times N_T \times L_D \times U_T \qquad (1)$$

where L_s is the additional load on the server process due to processing client transactions, N_C is the number of clients, N_T is the average number of transactions per client, L_D is the average dispatch load of a transaction request at the server, and U_T is the average client polling rate. Equation (1) is of course schematic. The *loads* L_S and L_D will be taken to refer to the number of CPU cycles devoted to the client-side transactions.

Note that 'throwing money and threads' at the problem does *not* reduce the load as defined above. Faster, multi-core computers are of course able to do more in a given time interval. Using a thread for each transaction can also reduce the impact of sluggish servers on the client side. But in the end, the total number of CPU cycles involved will be the same (if not more, due to extra thread synchronization and context switching).

Similarly, the average load on a server's network port is

$$L_N \sim N_c \times N_T \times P_T \times U_T \qquad (2)$$

where L_N is the network load (bytes per second), N_C, N_T and U_T are as before, and P_T is the average transaction payload. This does not depend on the number of threads used or the CPU power of the server.

A real reduction in load (server or network) involves reducing either N_C and N_T or both in the above equations. This can either be accomplished artificially (for instance by imposing restrictions on the number of and location of clients allowed to run and the update rates they are allowed to use) or moving to other data flow models.

1st Order: Contract-based Publish-Subscribe

As most control system data is used primarily in display at the client side, moving to an asynchronous publish-subscribe model can work wonders reducing the load on a server. Doing so eschews the 'keep it simple' approach, as connection and contract management are needed. A transaction request now results in a *contract* managed by the server, along with a table of attached clients. Nonetheless, the average load on a server due to client requests essentially becomes

$$L_S \sim N_T \times L_D \times U_T \qquad (3)$$

That is, the number of clients no longer plays a role. A transaction request is cached and made once on behalf of all N_C clients.

The outgoing network load (2) essentially remains the same, as the transaction results need to be passed to all interested parties. The incoming contribution to network load is for all practical purposes decimated, as transaction requests are made far less often. In order to further reduce the network load, one can adopt a 'send-on-change' policy, or reduce the number of clients by delivering data via multicast (especially effective for those transactions involving large payloads). The TINE control system protocol supports both of these features.

Asynchronous, publish-subscribe based protocols have a much larger domain of applicability, which however is still finite for several reasons. First, the API paradigm

still permits plaguing a server with an extra large number of transactions, N_T. Second, if application programmers have complete freedom in choosing their platforms and programming styles, client applications may still engage in synchronous polling, effectively reintroducing the N_C factor, and (depending on contract management) possibly imposing an additional asynchronous/synchronous coupling factor proportional to $(N_T)^2$.

To combat the latter two effects, one could restrict the available platforms to those *officially* approved, and to *police* the set of API standards. Or one could take steps to coerce efficient data acquisition at the protocol level. The TINE control system has now introduced many new second order hand-shaking features in this direction.

2nd Order: Contract-Negotiation

Client applications (and middle layer servers) require data from the control system for display and control of the machine. Specifically tailoring applications for efficient data transfer seldom enters into the picture. Indeed some APIs do not even offer this capability.

So on the one hand we have client applications driving control system data flow by making transaction requests (*contracts*), and on the other we have servers which bear the brunt of any ensuing scalability or efficiency problems. Servers are of course responsible for collecting the data and controlling the hardware. Thus, minimizing the impact of a server's data delivery plays a strong role regarding scalability.

Various strategies are available for reducing N_T and N_C in the above equations. In principle, one could use a purely *push* approach, where all of a server's available data are pushed via multicast onto the network. Although this might reduce the server load, it could drastically increase the overall load on the network. In addition it would require clients to sift through all data from a server in order to find the portion of interest (increasing client load). Nonetheless, pushing certain *popular* data elements (such as beam energy and current) is in general a good idea.

A server may also reduce the number of transactions it deals with if it can analyze the initial client request and, if possible, map it onto an existing contract, or anticipate further requests and appropriately restructure (*negotiate*) the contract request. We shall show below how this is done. In order to understand the principles involved, we present a brief review of control system API models.

We note that efforts to keep the dispatch load L_D to a minimum should in any case be made. The best practice involves simply copying ready data within the dispatch (rather than launching into numerical calculation or hardware readout).

CONTROL SYSTEM MODELS

Database Model

One can view the data flowing in a control system as deriving from elements in a database. This is the EPICS [3] approach, where one transfers *process variables* between the client and server. So the process variables have *names*, and the actions on the variables are one of *put*, *get*, or *monitor*.

Device Server Model

One can regard control system elements as controllable objects managed by a server. The *instance* of such an object is a *device*, which has a hierarchical name. The actions pertaining to the device are given by its *properties*. With minor differences in nomenclature and degree of object-orientation this is the model used in ACS [4], DOOCS [5], STARS [6], TANGO [7], and TINE.

Property Server Model

Certain control elements do not lend themselves well to a device oriented view but nonetheless follow the basic hierarchical naming scheme of the device server. This is typically true of middle layer services. Here one does not think of a *device* having *properties*, but of a *property* applying to different *keywords*. This model is also sometimes used in STARS and TINE, but is not available in TANGO or DOOCS.

TRANSACTION COERCION

Below we give some examples of transaction coercion and make frequent references to the *property* mentioned in the device server and property server models above, as this is the real focal point of the server transaction.

Multi-Channel Arrays

Client panels frequently attach individual elements of a *collection* to different display widgets, e.g. power supply controller (PSC) currents, beam position monitor (BPM) positions, or vacuum pressures. In large machines, this could amount to 100s if not 1000s of single elements.

TINE, however, allows a registered property to declare itself a *multi-channel array* (MCA), capable of delivering all elements of a given property as a vector (with a device order determined by the server). A rich client might directly request an MCA with all elements. Panels or strictly OO clients will not do this. However, contracts to obtain a single element of such properties are now renegotiated into a contract delivering the entire array. The client is informed via 2nd order handshaking as to the array index to device cross-reference. Thus a server only maintains a single MCA contract. The data arriving at the client is parcelled out into the individual single-element calls underneath the API. Recently, additional server side registration enables the specification of group devices, for cases where a property logically separates into sub-groups.

User-defined Types (Structures)

TINE also allows a server to define its own data types (*structures*) which a property can use in order to delivery a collection of data as an atomic unit.

Although a wonderful advent for rich client applications, structures present a display problem for

simple panel clients, which are more likely to request the structure fields individually. A server seeing such a request will deliver the entire structure, which will be re-packaged at the client. Again, many individual requests will collapse to a single contract managed by the server.

Collapsing Equivalent Contracts

In order to reduce the number of transactions it is important to make sure that equivalent calls collapse to the same contract. As aliases assume their canonical names when accessed via a client, they are unproblematic in this regard. However, a de-facto alias (device number instead of name) or an irregular array length or data type could result in a transaction occurring multiple times. Although possible to deal with via property registration, it is generally up to the server to reject non-standard requests with the appropriate error message.

Polling Intervals and Scheduling

Client applications sometimes need to know 'the moment something happens' and therefore request an update rate much faster than is otherwise necessary. A server can gracefully coerce such impatient clients to use a slower update rate by establishing a minimum polling interval. Once again, 2nd order hand-shaking renegotiates this with the client. A server can satisfy the needs of its clients by scheduling the requested property the moment there are new data to send, thereby reducing latency to essentially zero and obviating any need for fast polling.

Steering the Acquisition Mode

The payload delivered in some transactions can be very large (e.g. video frames or large traces). So even though the number of transactions might be at a minimum, the number of clients receiving the payload can result in a drain on network resources. The best practice here is to coerce all clients interested in large-payload transactions to use TINE multicast. A property can automatically renegotiate all asynchronous contracts to use multicast access (and reject synchronous requests), if so registered.

In a similar vein, properties can also reject synchronous calls in such a manner that an asynchronous listener is inserted under the synchronous call at the client side.

On the other hand, asynchronous monitoring makes no sense if the monitored data are static (do not change). An attempt to monitor such data will result in instructing the client layer to cancel the monitor.

Exclusive Read

A server can declare a property to have exclusive read characteristics, making it available only to those clients who pass through the same security screening applied to *write* transactions (*commands*). This can be used to allow time-consuming reads (e.g. extra large video signals) to be available only to a subset of the total client space.

RESULTS

Making use of these 2nd order techniques generally involves investing some time at the server front end, registering properties so that transaction coercions can take place. The benefits of doing this, however, can be dramatic. Some examples follow.

The FLASH magnet control consists of approximately 260 PSCs and is realized by various TINE servers (a primary server running on a Solaris host, and several PC104 servers running embedded linux). The client side applications are primarily DOOCS DDD [8] panels and MATLAB applications, all of which acquire settings and values from each PSC individually. Prior to introducing the techniques described above, the primary server had a constant background of ~1060 contracts, was being synchronously polled with > 500 contracts per second, and was at the high end of CPU usage. By introducing MCA access and static listeners for most of the synchronous polling, the number of background contracts is now ~ 50, there are much fewer synchronous calls, and the CPU usage is now back to 10 % or less. The client applications themselves were not modified in any way, other than relinking with the new libraries.

The mixed 100 Mbit/1Gbit infrastructure at PETRA3 introduces complications when delivering video images via multicast, especially if Gbit video servers or routers have 100Mbit video clients. As there is limited flow control, data delivery parameters must be precisely tuned. The most reliable performance was achieved by enforcing, via property registration, multicast access and a minimum polling interval.

The PETRA3 orbit server consists of ~270 Libera BPM readout modules which are attached to a single Linux CPU. Most properties are registered to provide MCA access. A minimal polling interval of 10 Hz holds the regular bevy of ~20 clients to a set of ~35 contracts and with a total CPU load of ~6 %.

We have shown in this report various methods whereby a server can take control over its clients. A server can continue to provide all callers with the information requested, but do so on its own terms.

REFERENCES

[1] http://tine.desy.de
[2] Duval et al., "TINE: An Integrated Control System for HERA", Proceedings, PCaPAC'99, 1999.
[3] http://www.aps.anl.gov/epics/
[4] http://www.cosylab.com/solutions/ICT/ACS/
[5] http://doocs.desy.de
[6] http://pfwww.kek.jp/stars/
[7] http://www.tango-controls.org
[8] http://jddd.desy.de/
[9] F.Schmidt-Foehre et al, "Control System Integration of the PETRA III BPM System based on Libera Brilliance", Proceedings, ICALEPCS 2009.