Server Node Reply Logic

Review analysis

Fri, Sep 17, 2004

The logic to support RETDAT server requests is somewhat obscure. This note is meant to clarify how it works.

Introduction

As an example of server node processing, consider the simple, but common, case of a RETDAT data request for a single device-property that specifies a reply return rate of 1 Hz. Imagine that a server node receives this request. Right away, the server node builds data structures to support the request until it is canceled. It then forwards the request to the target contributing node. That node sees the request as one for its own data, so it builds data structures to support the request and arranges for prompt generation of the first set of reply data and its delivery to the server node. When the server node processes the reply, it copies the reply data into the reply buffer that is part of one of its server support structures, and since it is the first reply, it delivers the first reply to the original requesting client. All this means that no time is wasted in delivering the first reply. (This is especially useful for one-shot requests, for which the first reply is the only reply.) Thereafter, every 15 cycles (at 15 Hz), another reply is generated and delivered by the contributing node early in its operating cycle, and the server node delivers the latest reply data at a time later in its own simultaneous operating cycle. To be specific, the server node might receive a reply from the contributing node at, say, 10 ms into the 66 ms cycle, and the server node would deliver that reply to the client at 40 ms into the same cycle.

Now suppose the contributing node "drops out," or fails to reply during the cycle expected. The server node reports this as a Tardy error, meaning that the expected data from the contributing node did not arrive in time for the reply to be delivered to the client relatively late in the operating cycle. After continuing failures to reply, the request is resent to the contributing node, hoping to remind it to participate in the request. The notion here is that the contributing node may have died and perhaps will soon be reset, after which a reminder of the data request will help it to again return suitable replies to the server node.

Complexities

The real world is often more complex than the simple case described here. A server request may include more than one device spread across more than one node, so that replies from multiple contributing nodes are anticipated. In this case, the request is forwarded to a multicast address that is designed to reach every other node that the server node could possibly be asked to serve. In this way, we avoid sending multiple copies of the same request to the various nodes affected. (The SSDN field in the RETDAT device packet identifies the node for each device.)

Any node may become a server node for any request that it receives, depending on what devices are in the request and how it received the request. The rules for deciding to act as a server node for a given request are these:

1. The request was received directly (not multicast)
2. At least one device in the request is non local

Passing these two tests causes the node to act as a server node for the request, forwarding it to the single non local node, or in the case of more than one non local node, multicasting it. Failing either of the two tests means it responds simply to the request, ignoring all devices therein except its own. Restated, a request that consists only of devices that are local is

responded to simply. A request that is received via multicast is also responded to simply. A request that is received via multicast is ignored if it contains no local devices.

In order to support requests that include both local and non local devices, it is important that a multicast request be received by the local node; *i.e.*, it should be able to see its own multicast messages. This means that it must be a member of that same multicast group. When it sees its own server multicast request, it will again examine the same request, but this time it will "wear a different hat," since it will determine not to be a server node.

There is a complication when resending a request. If the original request was forwarded to a multicast destination, then a resend must be sent only to the node from whom a reply is missing. But to send such a request to a single node, one must take care not to include any devices that are not sought from that node. Failure to pare down the original request to one whose devices only target that node will cause the node to think it should be a server for that request, which would be definitely undesirable.

To organize all this, we normally select a multicast address that is used by all the nodes in one project, say, all Linac front ends. On the Acnet side, what determines whether a device will need support via a server node is whether its "source node" is the same as the node that is part of the `SSDN` for that Acnet device-property. In this way, the nodes of one project will not see the multicast requests of another project.

Synchronization

As mentioned earlier, it is important that all nodes in a project operate synchronously, each using a "micro-p start" interrupt that is delayed equally from the same clock event. A server node can therefore assume that each contributing node should have returned its replies early in the cycle compared to the relatively late time in the cycle (40 ms) that `ACUpServ` is called. This is because each node executes its `Update` task at the start of each cycle to perform three key jobs, seeing that its own data pool is updated with fresh data, all local applications are given a chance to run, and all simple replies that are due on that cycle are updated and delivered. An active page application runs only after the `Update` task is done. If the `Update` task does not complete soon enough so that all replies can be delivered to a requesting server node by 40 ms into the cycle, then the node is not properly configured.

Another element of synchronization is that the front end processes all active requests at the same time in the cycle. Nonserver requests are processed early; server requests are processed late. Since there may be many reply messages to deliver to a single requesting node, perhaps with different values of the `ACFTD` field, it is more efficient to combine the replies, if possible, into a single datagram. To that end, when the network message queue is flushed, messages targeting the same destination are combined as much as possible. In order to make it easy to recognize this situation, the linked list of active requests is maintained in destination order, so that processing all active requests might queue consecutive multiple replies to a common destination. On the other end of this, processing of received Acnet datagrams must be ready to separate out multiple messages. Each concatenated message includes its own Acnet header in order to simplify the required handling.

The exception to processing all active requests at the same time in a cycle is the handling of a request when it first arrives. The request is initialized right away, and if server node support is needed, it is forwarded right away. As the first immediate replies from the contributing nodes are received, a check is made for a complete set. As soon as the last contributing node delivers its immediate first reply, `ACUpServ` is called to deliver the full reply to the original client. Since this logic can happen anytime during the cycle, such replies are unlikely to be

concatenated. But one-shot requests thereby result in very prompt replies.

Logic organization

Nearly all of the code relating to server node replies is in `ACUpServ` in the `ACReq` module. As part of a server request structure, the `FTDCNTR` is key, working with the `ACFTD` field of the `RETDAT` request to support periodic replies. (The `ACFTD` is the 16-bit "frequency time descriptor" field from the original request. If it is positive, it represents a reply period in units of 60 Hz cycles. If it is negative, the low 8 bits specify a clock event number, where each occurrence of that clock event should prompt a reply.) During `Server` task processing, usually scheduled at 40 ms into each 15 Hz cycle, `ACUpServ` is called for each active `RETDAT` request. During initialization of a periodic request, the `FTDCNTR` field is initialized to 2 or 3, depending on whether the request arrived before or after the `Server` task executes. Every time a reply is delivered, `FTDCNTR` is reset to the period of the request expressed in cycles. Upon entry to `ACUpServ`, this field is decremented each cycle, and when it reaches zero, it is time for a new reply to be delivered to the client. This allows at least 2 full cycles for the first time replies from all contributing nodes before declaring the default NoResponse error of "`36 –8`". But the logic in `SReply`, which processes each contributing node reply, notices when all first time replies have come in, after which it calls `ACUpServ` to deliver the first time complete reply immediately.

One-shot requests also initialize `FTDCNTR` to 2 or 3, just the same as for the periodic case. After delivering the only reply, the `FTDCNTR` is set to –1, and no further processing occurs.

The clock event case is different, and `FTDCNTR` is always set to –1. An update occurs on every cycle for which the clock event is true.

The subtle part of the logic is in the preparation for replying, as the reliability of the replies from each contributing node must be monitored. To that end, the `SAGE` field of each device in the request is used. It has two basic forms that are distinguished by the sign bit of the 16-bit word. If the sign bit is zero, the low 15 bits are a time stamp that is the cycle counter of the last received contributing node reply. If the sign bit is one, the low 14 bits are a time stamp of the cycle counter of the last time a reminder request message (resend) was sent as a result of determining that replies are missing for too long. The other bit (#14) in this second form is a flag to denote two other special values. One is the value `0xC000`, which is used to initialize the `SAGE` field. The other is `0xFFFF`, which is used for the `SETSERV` case, which supports acknowledgment replies for a `SETDAT` message that needs server node support. This `0xFFFF` value prevents any resends.

Once we enter the updating part of `ACUpServ`, for a request that is not a one-shot, each device is scanned in turn and the `SAGE` field monitored. Let's look at the case of the first time this logic is entered. If the `SAGE` field is `0xC000`, it means that no contributing node reply has yet been received for this node, so a resend is sent to the associated contributing node, with the request pared down to include only that node's devices. If a reply had been received, the logic in `SReply` would have changed the `SAGE` field to a positive value, with the low 15 bits the time stamp of that reply's arrival. Once a resend is sent, the `SAGE` field is changed to `0x8000` plus the low 14 bits of the current cycle counter. The next time a reply is consistently missing, we will see the sign bit set again, and only if sufficient time has passed since the last resend will we repeat the resend.

When the `SAGE` value is positive, it means that a reply that was last received has its time stamp in the low 15 bits. Comparing that time stamp with the current cycle counter, we decide whether the reply was recent enough; if it was not, we mark the status word

associated with that device's data to Tardy, or "`36 –7`". Note that as the loop progresses through all of the devices, each one from that same node will be so marked, since the time stamps all match. There is no immediate resend taking place just because of this, but the `SAGE` is set negative with the low 14 bits of the current cycle as a time stamp. The next time an update is due, this time stamp will be compared with the then current cycle counter, and if enough time has passed, a resend will be issued. All this means that a resend is not an immediate reaction to a missing reply from a contributing node, but declaring missing/tardy status is. One way to issue the resend more promptly would be, upon detecting missing data, to set the negative `SAGE` value to have a time stamp in the past compared to the present cycle counter. Then, the next time that missing data is detected, a resend might be delivered right away. But in either case, the resend is not an immediate response to missing data.

The clock event case is similar. Since the server node is aware of the occurrence of clock events, it can detect when a reply is missing. The time stamp of the reply should match the present cycle counter. An exception might be allowed for the first immediate reply, in the case that the original request arrived at the server node near the end of the cycle. In that case, the reply could have come also in that same cycle, but waiting for the other nodes to reply may have pushed beyond the cycle boundary, so that by the time a check is made, the reply of one or more nodes could seem to be one cycle late. So we need to be a bit flexible in judging how recent the reply was.

When a resend is issued, it is important to be sure that any other devices whose data is expected from the same node should have their corresponding `SAGE` fields set to the same value, that of `0x8000` plus the low 14 bits of the current cycle counter. As those devices are then encountered in the same loop, they will not prompt a redundant resend.

As for the status field associated with each device's data, it is set to NoResponse, or "`36 –8`", when the request is initialized. This default value means that no reply has ever been received for that device. Every time a reply is received, `SReply` copies the status word from that reply into the reply buffer it builds for the original client, overwriting the default value. Tardy data is only detected after at least one reply has been received from a contributing node.

Logic flow in pseudo code

Here is pseudo code for deciding whether an active server request should be updated. This logic is in the early part of `ACUpServ`.

```
update = false
if FTDCNTR > 0
        decrement FTDCNTR
        if FTDCNTR = 0
                update = true
else if FTDCNTR < 0
        if ACFTD < 0
                if clock event true
                        update = true
else if FTDCNTR = 0
        update = true
```

Note that if `FTDCNTR` = 0 on entry, the request will be updated. This is the case for the call from `SReply` when the last of the first time replies has been received. Also note that no update occurs for a negative `FTDCNTR` value except for the clock event case. The `FTDCNTR` is set to –1 after the reply to a one-shot request is delivered.

The next flow is done in case the update flag is true and the request is not a one-shot. Outside of the loop over all devices in the request, we have the following:

```
if ACFTD > 0
        recent = ACFTD in units of cycles
else
        recent = 0

if reqUpdateCnt = 0
        increment recent
```

Note that the last test allows an extra cycle of patience for the first reply, allowing that this might be happening very near the end of a cycle.

The following code is inside a loop over each device in the request.

```
if SAGE >= 0
        time = (cycleCnt – SAGE) & 0x7FFF
        if time > recent
                status = Tardy
                SAGE = 0x8000 + (cycleCnt & 0x3FFF)
else if SAGE != –1
        if SAGE = 0xC000 OR ((cycleCnt – SAGE) & 0x3FFF) > 30
                resend request
                SAGE = 0x8000 + (cycleCnt & 0x3FFF)
                propagate same value to all other SAGE fields of this node
```

The `cycleCnt` global variable is incremented by the system code at the start of every cycle. Note that nothing happens in the above logic if `SAGE` = `0xFFFF`. Also, this logic prevents resends from happening more often than every 2 seconds (30 cycles). Note, too, that a resend cannot happen on the same cycle on which a Tardy status is marked; a resend has to wait until at least 2 seconds later, allowing more chances to receive a reply before giving up and issuing a resend. Following execution of all the above, the reply buffer is delivered to the client. The `reqUpdateCnt` field of the server request block structure is also incremented.

In the `SReply` code, which is called to process a reply from a contributing node, we have this logic inside a loop over the devices that stem from that node:

```
if SAGE < 0
        increment firstTimeCnt
SAGE = (cycleCnt & 0x7FFF)
```

Note that the check on the `SAGE` value insures that we only count first-time replies of each device's data. This works because the `SAGE` field is set to `0xC000` at request initialization.

After the above loop in `SReply`, we have the following:

```
if reqUpdateCnt = 0
        if firstTimeCnt = deviceCnt
                FTDCNTR = 0
                ACUpServ
```

Thus, as soon as the first set of replies is received, the complete first reply is delivered.

Server reply status

The reply data for each Acnet device in a request is preceded by a status word. As a server request is initialized, its status word is preset to NoResponse, or "`36 –8`". If any contributing node is down, say, so that it cannot contribute, the server node's reply will thereby include this status for each of its devices. If there is a reply, the status word in the contributing node's reply is copied into the server node's reply buffer along with the data. But if the reply is not recent enough, it is overwritten with the Tardy "`36 –7`" status described above. The only nonzero value of status coming from a contributing node is BusError, or "`36 –4`". For any other possible error detected during initial request processing, a status-only reply is returned to the client, and the request is rejected.

Summary

This note describes how server node logic works during `RETDAT` request processing. The logic replies upon synchronized cyclic operation across all nodes in a project. Missing replies are marked with Tardy status. A persistent series of missing replies from a contributing node is regarded as evidence that a node is down, and a resend is issued for that node as a means of reminding it, once it comes up again, of its obligation to participate in an active request.

Postscript

The careful reader may note that the loop over each requested device in the `ACUpServ` logic is a bit more lenient than is necessary. Consider a 15 Hz request, for which a value of the `ACFTD` field of `0x0004` specifies that replies should be returned every cycle. Except for the first update, the "recent" variable above is 1. The check of the "time" variable against "recent" allows for one more cycle than is necessary, since in this case, reply data should have been received on the current cycle, so that "time" would be 0. In fact, a single missing reply in this case would not be noticed. Once stable operation without errors is achieved, we might change the comparison operator to from > to >= as follows:

```
if time  >=  recent
```

Again, for this 15 Hz case, the "time" variable would be 0, and "recent" would be 1. A single missing reply would mean "time" is 1, resulting in Tardy status reported.

The main reason for a server node's monitoring of contributing node replies is to cover the case of a node that dies and subsequently comes back to life. A secondary reason is to report that reply data is Tardy, or stale. It is expected that the network is reliable, even based upon UDP protocols, as so much else in our entire control system depends on its reliability.