

Queuing States

GETS32 States Support

Thu, Oct 18, 2007

States support for the `GETS32` protocol includes a States Pool Table that is referenced when fulfilling data requests that are based upon a State variable. When a State announcement is received, the corresponding SPT entry is updated to capture that announcement. As with all data request support, requests are fulfilled only once per 15 Hz cycle. That raises the question, what if two State announcements, for the same State variable, are received within a single 15 Hz cycle? Might the first announcement be missed? This note explores how this is handled by modifying how the SPT is used by the `STTE` local application that processes State announcements.

Each SPT entry includes, besides the State announcement data, a byte counter that is incremented whenever the SPT entry is updated. Each 15 Hz cycle, at the time of request fulfillment, each State-based request checks whether the counter has changed since the last cycle. If it has, it knows that an announcement has occurred, and it can determine whether a reply is due this cycle. That logic only retains the low 4 bits of the counter to detect when a new update has occurred.

The scheme described here manages updates of the SPT so that any given entry is updated no more than once per 15 Hz cycle. It does this with the help of a queue that holds waiting entries.

The number of queued entries, including the current entry, is housed in a `qCnt` byte. This meaning applies before the `ReplyDue` calls are made during data request fulfillment by the `Update` task.

Any local application is given a `cycle` call every 15 Hz cycle shortly before request fulfillment. This is the logical time to dequeue any waiting State announcements to matching SPT entries, in order to prepare the SPT for the `ReplyDue` calls. Now, what is currently in the SPT entries was seen by `ReplyDue` last cycle. After that, the entries are stale; they carry no new information for `ReplyDue` to examine, because the counter byte has not changed. For each SPT entry with a `qCnt` field > 1 , check the queue for the oldest matching State announcement, if any, dequeue it and place it into the SPT entry, decrementing the queue counter byte.

Now consider a refinement for the sake of efficiency. In each `net` call, when a State announcement message is processed, update the matching SPT entry as appropriate, rather than always queuing the new entry. But before doing this, dequeue any waiting State announcements. The logical time to do this is after request fulfillment. But it may be done any time before the next `net` call, or in the absence of same, by the next `cycle` call. To enable this logic, use an internal counter that is cleared at the end of each `cycle` call and incremented at the end of each `net` call. Then either call can check the counter at first, and if it is zero, perform the needed dequeuing.

The dequeuing checks the `qCnt` byte. If it is 0, do nothing. If it is 1, there are no more queued entries, so decrement it to 0. If it is 2 or more, copy the oldest queued matching State record into the SPT entry, mark the queued entry as used, and decrement `qCnt` in the SPT entry, leaving it as 1 or more. This process loops through all SPT entries, but if we maintain a list of all SPT entries that have nonzero `qCnt` fields, it can be speeded up.

Again, to process each State announcement, check the `qCnt` byte. If it is 0, do nothing. If it is 1 or more, the entry is already holding a State announcement, possibly one that was just dequeued, so append the new one to the queue and increment `qCnt`.

Functional tour

The functions relevant to the above logic are these, including the argument types:

```

Queue(StateRecPtr)
DoState(StateRecPtr)
QPrune
QFind(DevIndex)
Dequeue
DoNet

```

The `DoNet` function processes a multicast States announcement message that includes one or more `StateRec` structures. After checking for a valid message, it calls `DoState` for each State announcement included in the message. It also logs any cases of received messages for which the sequence number field does not increment by 1. This relates to multicast reception reliability.

The `DoState` function either updates the matching SPT entry or queues it to be handled later. It decides this based on an argument and the `qCnt` field. If `qCnt` is 0, the entry is vacant, so it updates the entry with this announcement and sets `qCnt = 1`. But if `qCnt` is nonzero, the entry is already updated for this cycle, so it calls `Queue` and increments `qCnt`.

The `Queue` function merely adds a State record to a simple queue structure.

The `Dequeue` function is called once per cycle to “shift out” the oldest queued entry for each State variable into the corresponding SPT entry. Logically, this action should be done following request fulfillment time but before any new State announcements are received. It is done at the start of the first `net` call processing, or if no messages are received, then at the time of the `cycle` call to `STTE`. For each SPT entry that has a nonzero `qCnt`, it decrements `qCnt` and installs the oldest queued matching State announcement, if any, to that entry, marking each removed queue entry as used. (If `qCnt = 1`, there are no corresponding entries in the queue, so it merely decrements `qCnt` to 0.) To do this work efficiently, since the entire SPT may have hundreds of entries, a list of SPT entry pointers with nonzero `qCnt` fields is maintained. So `Dequeue` merely works through that list and makes the check on each such SPT entry. As it does so, it builds an updated list.

The `QFind` function merely finds a match by examining each queue entry beginning with the oldest. It is called when `Dequeue` finds an SPT entry with `qCnt > 1`. When `Dequeue` needs to update an SPT entry from the queue, it calls `DoState`, with an argument that disables queuing. After the dequeuing is done, it calls `QPrune`.

The job of `QPrune` is merely to advance the `OUT` index toward the queue’s `IN` index, skipping past any used entries that have already been removed from the queue to be installed in an SPT entry.

As of this writing, there are a few limits in the implementation:

<i>Name</i>	<i>Value</i>	<i>Meaning</i>
<code>MAXDIFFS</code>	93	max #entries in circular buffer of unusual consecutive sequence#s
<code>MAXREC</code>	127	max #state records in one multicast States announcement message
<code>MAXQUEUE</code>	255	#entries in circular queue
<code>MAXNZ</code>	116	max #entries in <code>nzList</code> of SPT entries with nonzero <code>qCnt</code> fields

In operation, during testing, it seems that these limits should not be a problem. But if they are, it should be easy to modify them and build a new version of `STTE`. Multicast messages are apparently on occasion found missing when `STTE` is run in a test node. It seems much better in an operational node. The #state records in a message is only occasionally more than 1 or 2. The #entries in the queue is usually not more than 1. The max size of `nzList` may be only 13 or so.