Critique 3

The paper is well written, which addresses how to implement middleware level support for aperiodic tasks. I find the descriptions of deferrable server mechanisms and admission control are particularly interesting. For deferrable server mechanism, there are multiple threads with different priorities. The budget manager makes all the scheduling while the server thread executes the aperiodic tasks. However, this design heavily relies on the priorities of each thread. If there is any priority inversion among any threads, the scheduling policy will fail. By making system calls to the server thread, the budget manager acts as the headquarter and frequently intervene the server thread. I am concerned with the overhead of this synchronization, which is also stated in the paper. Yet, as there is only one mutex globally and the budget manager is the default holder of the lock, I think the overhead can be greatly reduced. Of course, another good way is to use shared memory to facilitate the communication among threads. However, this approach might also have significant fallback as it might take considerable amount of memory and need to be handled carefully. For the admission control service, I think it is inspiring as I have never thought to have an additional process to monitor the incoming tasks. The door keeper process is responsible for accepting or rejecting tasks based on current schedule. As mentioned in the paragraph, I think the policy of ejecting noncritical tasks to accommodate critical ones need to be further extended. While randomly picking one to reject is rather easy, the system utility might be compromised. Moreover, I am concerned with the overhead of this handshaking process. While AP and AC are deployed together in a single host (or VM), applying lightweight interprocess communication such as unix domain socket and shared memory would greatly reduce latency. If these two processes are on separate hosts, the round trip latency is somehow out of control as bandwidth and underlying TCP/UDP socket buffer are unknown. Another thing that is noteworthy is the implementation of idle detector thread to report finished aperiodic tasks. However, the detector has the lowest priority and can only be active when there is a processor idle. In a CPU contention case (all processors are flooded with tasks), the detector might never get scheduled, thus its designed function is compromised. In the Section 4, there is a detail breakdown of the latency for each individual step. The communication between process is the longest as expected. Yet, we also find a moderately long latency for sending idle events and sending “accept” query. While they are relatively large compared to other steps, I think it is the overhead that we must sacrifice to achieve the door keeper function. The aggregated admission controller makes the scheduling mechanism more modular and more manageable.