# Hard-real-time Resource Management for Autonomous Spacecraft[1]

Erann Gat
Jet Propulsion Laboratory, California Institute of Technology
4800 Oak Grove Drive, Pasadena, CA 91109
gat@jpl.nasa.gov

*Abstract*— This paper describes *tickets*, a computational mechanism for hard-real-time autonomous resource management. Autonomous spacecraft control can be considered abstractly as a computational process whose outputs are spacecraft commands. In order to make the engineering of such systems tractable, these computations are usually organized as multiple parallel threads of control. It is sometimes necessary, particularly in emergency situations, for one thread of control not only to issue certain commands, but to actively prevent the issuing of other commands by other threads of control. Tickets are software objects that act as intermediaries between control processes and low-level commands. In order to gain access to a low-level command a control process must be in posession of a valid ticket for that command. Tickets can be invalidated in constant time. This allows hard-real-time guarantees of performance for denying access to a particular low-level command or set of commands.

TABLE OF CONTENTS

1. INTRODUCTION
2. TICKETS
3. CONSTRAINED TICKETS
4. SUMMARY AND CONCLUSIONS

## 1. INTRODUCTION

A *resource* is anything that causes dependencies among components of system state. For example, electrical power is a resource that causes dependencies among device power states: the amount of power available constrains the combinations of device power states that are physically achievable.

Failure to honor resource constraints can result in loss of spacecraft, so resource management is a critical part of any spacecraft control architecture. Traditionally, resource management has been done on the ground through careful modelling of the effects on system state of open-loop time-based command sequences. But this approach is expensive, and does not support new mission concepts that require autonomous operation.

Resource management in general is a very complicated problem that has an extensive literature, e.g. [1]. Most work in the field has focused on resource management through planning and scheduling in advance of actual performance. In this paper we focus on a narrow and largely ignored aspect of the problem: hard-real-time resource management for autonomous reactive responses to contingencies in the context of a control system consisting of a number of independent parallel threads of control. This situation arises in state-of-the-art autonomous spacecraft control systems such as the JPL Mission Data System (MDS) [2]. (There are simpler solutions to the problem that are applicable in cases where execution control is centralized, as in the ARC/JPL Remote Agent (RA) [3]).

To motivate our solution, consider the following example: a spacecraft is performing a science observation using a camera when a thruster valve sticks open and the spacecraft starts to spin up. The only way to shut off the flow of propellant to the thruster is to activate a cutoff valve, which has a transient power draw higher than the current power margin. Attempting to activate the cutoff valve with an inadequate power margin will result in a bus trip and loss of spacecraft. The only way to save the spacecraft is to turn off the power to the camera and *make sure it stays off* while actuating the cutoff valve. This has to be done quickly or the spacecraft will spin up to unrecoverable rates.

It turns out that solving this problem while meeting all the stipulated conditions is quite tricky. This is somewhat counterintuitive, so it is useful to start with some obvious solutions and describe why they don't work.

### *Things that don't work*

The simplest solution is to simply have the system issue the commands to turn off the camera and then actuate the cutoff valve. This solution fails because of the stipulation that the control system is multi-threaded. There is no guarantee that some other thread of control will not turn the camera back on between the time when it is turned off and when the cutoff valve is actuated, which would result in loss of spacecraft. In fact this situation is not at all unlikely. It is common to assign threads of control the responsibility for maintaining goal conditions in the system, so there would be a thread whose responsibility it is to make sure that the camera remains on for some specified period of time. For example, in the MDS architecture the camera would likely

[1] 0-7803-5846-5/00/$10.00 © 2000 IEEE

have been turned on in service of a *goal* [2] to maintain the camera power on for the duration of an observation. If the camera were to suddenly be turned off, the thread responsible for maintaining the camera power goal would try to turn the camera back on, resulting in a race condition (between the camera-on command and the thruster cutoff valve actuation command) with potentially fatal consequences.

A possible patch to this solution is to try to insure that the thread responsible for keeping the camera on is simply prevented from running until the cutoff valve is actuated. However, there are no guarantees about how long this thread will have to be disabled (since turning off the camera could fail due to transient errors), and it might be performing time-critical tasks in addition to keeping the camera on. This solution could work in special cases, but not in general.
A second solution is to explicitly manage the resource responsible for the dangerous state coupling, in this case electrical power. For example, we could require that the camera-power thread request and receive an allocation for power before turning on the camera, e.g.:

```
PowerAllocation A =
     getPowerAllocation(cameraPowerDraw);
if (A != NULL) turnCameraPowerOn();
```

The first problem with this approach is that its reliability depends *entirely* on a coding convention; nothing actually prevents a thread from turning the camera on without first obtaining a power allocation. A single violation of this convention can result in loss of spacecraft. And the convention is not nearly as simple as the above example implies. Simply preceding every call to turnCameraPowerOn with a call to getPowerAllocation will not work. For example, we might want to call turnCameraPowerOn to turn the camera back on after an SEU turned it off. In such a case we (presumably) already have an allocation, and we don't need to get another one:

```
PowerAllocation cameraPwrAlloc;

while (1) {
  Event e = getNextEvent();
  case (e.type) {
    ...
    // Turn camera on
    if (!cameraPwrAlloc.valid())
      // Only need this if we don't have
      // an allocation already
      cameraPwrAlloc =
     getPowerAllocation(cameraPowerDraw);
    if (cameraPwrAlloc.valid())
      turnCameraPowerOn();
    else
      // handle failed allocation
    ...
}
```

We also have to remember to give the allocation back when we're done with it:

```
    // Inside the event loop
    ...
    // Turn camera off
    turnCameraPowerOff();
    cameraPwrAlloc.release();
    ...
```

Even this is not yet enough. Calling turnCameraPowerOff does not guarantee that the camera is actually off, since faults can prevent the command from working:

```
    // Inside the event loop
    ...
    // Turn camera off
    turnCameraPowerOff();
    // Wait for effects to trickle
    // through state determination
    if (cameraPowerState.currentEstimate == OFF)
      cameraPwrAlloc.release();
    else
      // handle fault case
    ...
```

And even this is still not correct code! All of the conditionals need to run as critical sections, otherwise an allocation could be withdrawn between its validity check and the issuing of a command that relies on the allocation being valid.

Our simple coding convention is no longer so simple, and we should feel some discomfort gambling the life of our spacecraft on this convention being followed flawlessly throughout the flight code. The critical-section requirements are particularly pernicious, as the requirements are not intuitively obvious, and the bugs introduced by failing to adhere to them are intermittent, rare, and highly dependent on subtle differences in timing. It is possible to have such a bug that never manifests itself during testing suddenly appear in flight.

A better solution would be to encapsulate the requirements in an object class that is part of our programming infrastructure. That way we only have to insure the correctness of the infrastructure implementation, and not every individual use. This would reduce verification costs while making the code more reliable. In the next section we describe such an encapsulation.

## 2. TICKETS

We have developed a run-time resource management mechanism called a *ticket*. Tickets are software objects that act as brokers for access to low-level commands. They provide two major advantages over traditional resource management techniques. They do not rely on progammers adhering to a complex coding convention, and they provide guaranteed hard-real-time revocation of resources in emergency situations. Tickets can also be used to manage real-time shared resources, e.g. power sharing between thruster valves and catalyst bed heaters, and consumable resources like energy and propellant.

Tickets are based on a fundamental change in the way resources are viewed. Instead of viewing *states* like power margin as resources, tickets view *commands* as resources. Recall that a resource is defined as anything that causes

dependencies among components of system state and thus leads to potential goal conflicts. But effects usually have multiple causes, any of which could be considered a resource. Physics has no preference for one cause over another, making this a design choice.

For example, consider the situation where a device is turned on with insufficient power margin on the power bus resulting in a bus trip. There are two factors that together produce the bus trip: 1) a power margin below some threshold and 2) the execution of the power-on command. Both of these are necessary conditions for the bus trip, and so either one of them can be considered a resource according to our definition. Choosing power as the resource is useful when humans are in the loop planning command sequences, but as pointed out in the previous section it presents difficult problems for autonomous systems.

A control thread using tickets doesn't request access to *power*, but rather access to power *commands*. This access is granted in the form of a ticket object, which has methods associated with it that invoke the actual primitive commands, to which threads have no direct access. Primitive invocation is subject to the state of a boolean private attribute of the ticket called the *validity flag*. If the validity flag is false then ticket invocation fails. Reclaiming a resource therefore becomes a simple matter of setting the validity flag to false, a constant-time operation. Because threads have no access to primitives except through tickets and tickets enforce their validity flags internally this guarantees that resources cannot be accessed unless they are allocated.

*Ticket agents*

Tickets are granted by *ticket agent* objects. There is a ticket agent for every group of primitive commands for which a ticket may be granted. How this grouping is done is a design decision. For exampe, there may be a single ticket agent for all power-related commands, or there may be a separate ticket agent for the power commands for each device.

A ticket agent grants tickets according to a ticket *policy*. The simplest policy is to grant only a single valid ticket at any one time to the requestor with the highest priority. If more than one requestor has the highest priority then tickets are granted according to some heuristic like first-come-first-served.

This simple policy is sufficient to solve the stuck-thruster emergency described in section 1. In this example there are two threads of control, a camera monitor thread whose job it is to keep the camera on for an observation, and an emergency thruster cutoff thread whose job it is to actuate the thruster cutoff valve when the thruster sticks open.

There are several ways to organize the ticket agents for this example. The most straightforward is to create a single ticket agent object for both the truster cutoff valve actuation command and the camera power on/off command(s), and to make this object also be responsible for controlling the power margin state.

The example works as follows: first, the camera monitor thread requests a ticket for the camera power commands. Since there are no outstanding tickets for these commands the request is granted and the camera monitor receives a valid ticket. The camera monitor then uses this ticket to turn on the camera.

When the thruster sticks open we again have several design alternatives. The first is for the emergency thruster cutoff thread to request a ticket for the camera power commands so that it can turn the camera off to make power available for the cutoff valve actuation. This request has a higher priority (spacecraft emergency) than the camera monitor thread had (science observation) so the ticket agent invalidates the camera monitor thread's ticket and issues a new ticket to the thruster cutoff thread. The thruster cutoff thread uses this ticket to turn off the camera, after which it actuates the cutoff valve.

The camera monitor thread has no knowledge of the thruster emergency (it has no knowledge of thrusters at all) so it will perceive the camera's turning off as an anomaly from which it will try to recovery by attempting to turn the camera back on. However, the camera monitor's ticket has been invalidated so these attempts will have no effect. This eliminates the dangerous race condition that would have existed without tickets.

Once the emergency thread has finished actuating the cutoff valve, it releases its ticket for the camera commands (since it no longer needs to control the camera power state). When this happens, the camera monitor's ticket is once again the highest priority ticket, and so it is re-validated by the ticket agent. The camera monitor is now able to turn the camera back on. If circumstances permit (e.g. no deadlines have passed, attitude control is still possible) the observation could be restarted at this point.

It is easily demonstrated that all of these processes are constant-time operations. The list of outstanding tickets is kept sorted according to priority. There is only one valid ticket at any givn time, and it is always at the front of the list.

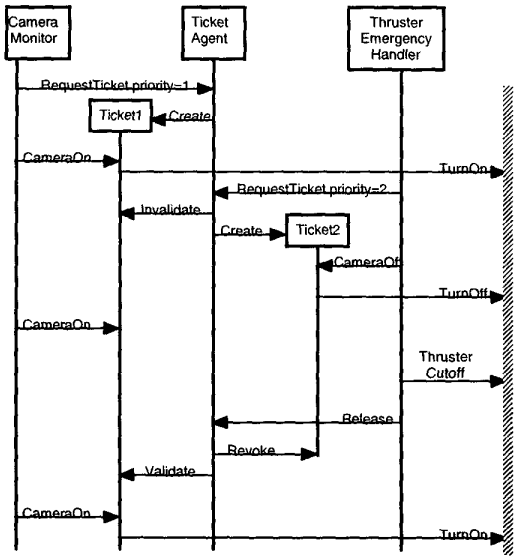


Figure 1: UML sequence diagram for simple prioritized ticket granting policy.

*A more realistic policy*

The scenario just described is somewhat unrealistic because the thruster emergency thread would generally not be aware that the camera needs to be turned off before actuating the thruster cutoff valve. The emergency thread would simply request a ticket for the cutoff valve actuation command and rely on the ticket agent to take whatever actions would be needed to be able to issue the ticket safely.

In this particular case this is straightforward because we stipulated earlier that a single object is the ticket agent for both the camera and thruster commands, as well as being responsible for managing the power margin. This object's ticket-granting policy would be to issue as many tickets as possible in priority order under the current power constraints.

In our example, the camera monitor would request and receive a ticket for the camera power commands as before. The ticket agent would internally bookkeep a power allocation for this ticket, since the camera can now be turned on at any time. Then the thruster emergency thread would request a ticket for the cutoff valve actuation command. There is not enough power left to satisfy this request, so the ticket agent searches for lower-priority allocations that can be usurped. In this case it finds one: the camera power allocation. This allocation can be made available by invalidating the ticket *and turning the camera off*. Note that simply invalidating the ticket is not enough. There is no inherent connection between a ticket and the actual state of the system. Tickets only constrain the *possible* states of the system. The actual state is determined by the objects that issue actual commands through the tickets they hold.

Note that tickets do not preclude more traditional resource allocation bookkeeping. Tickets are simply a mechanism of enforcing the connection between allocations and actions that actually consume the allocated resources.

## 3. CONSTRAINED TICKETS

In all the examples so far we have considered only electrical power as a resource. Power is easy to deal with because it is non-consumable; the amount of resource used at time T is a *function* of system state at time T. But many resources are *consumables*, where the amount of resource used at time T is an *integral* over system state through time T.

For example, suppose the spacecraft were running on battery power and an object (an observation manager, say) requests a ticket for the camera-on command. The ticket agent might want to enforce a limitation on the total time that the camera is on in order to avoid depleting the battery.

One possible way to do this is for the ticket agent to monitor how long the camera has been on and to invalidate the ticket (and turn the camera off) after the time limit is reached. But this is a poor solution because it will disrupt the observation, and whatever energy was consumed by the camera will have been wasted.

A solution to this problem is for the observation manager to pass as an argument along with the ticket request the total time that the camera is needed. The ticket agent can then determine ahead of time whether enough energy is available to satisfy the entire request before issuing the ticket.

There is still a problem: any time limit associated with the ticket still needs to be enforced, and at the moment it is the burden of the ticket agent to enforce the time limit by invalidating the ticket and issuing whatever cleanup commands are needed (e.g. turn camera off). But the ticket agent can't start the timer when the ticket is issued; it has to wait until the ticket is actually used, that is, when the command to turn on the camera is actually issued. But the ticket agent has no access to this information. Commands sent through a ticket go directly to the hardware, not to the ticket agent.

Again there are several possible solutions to this problem. One is to make the ticket send copies of all commands issued through the ticket back to the ticket agent that issued the ticket so that the ticket agent can track resource consumption. A second possibility is to build this capability directly in to the ticket itself. We choose this second design for reasons of modularity

To implement this solution, ticket validity is no longer a simple boolean but is instead a function that is called with

every ticket invocation. The ticket's command is issued IFF its validity function returns true on that invocation. The validity function keeps track of resources used by the ticket's commands, and enforces any restrictions, like time limits, that there might be. In order to keep the mechanism real-time this function must run in constant-bounded time. To enforce this, a fixed library of ticket validity functions is provided, all of which have been statically verified to run in constant-bounded time.

Ticket validity functions allow sophisticated management of consumable resources, as well as advanced real-time non-consumable resource management like electrical power sharing between thruster valves and catalyst bed heaters. For example, a ticket can be issued for thruster-firing commands whose validity function allows those commands to be issued only when the catalyst bed heater is off.

Because the ticket validity function may contain internal state (like integrators) and must be constructed on the fly, they are most easily implemented in languages that have first-class lexical closures (functions with internal state) like Lisp or ML. It is also possible to implement them in more impoverished languages like C++ by having a separate ticket sub-class for each possible validity function.

## 4. Summary and Conclusions

This paper has described *tickets*, a computational mechanism for hard-real-time resource management on autonomous spacecraft and other mission-critical embedded autonomous sytems. Tickets encapsulate the complex control structures associated with real-time resource management into an object class, making software more reliable and easier to verify.

Tickets are based on viewing commands rather than states as resources. States (like power margin) can still be bookkept as resources internally by ticket agent objects, but they are no longer considered resources architecturally.

Because tickets work by regulating access to primitive commands it is necessary that control threads have no direct access to those commands. Tickets also impose a computational overhead on control processes, but this overhead is constant-bounded, so tickets are suitable for hard-real-time applications.

## Acknowledgements

This work was performed at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

## References

[1] Steve Chien, et al. "Autonomous Planning and Scheduling for Goal-Based Autonomous Spacecraft." *IEEE Intelligent Systems*, September/October 1998.

[2] Dan Dvorak and Robert Rasmussen. "Software Architecture Themes in JPL's Mission Data System." *Proceedings of the IEEE Aerospace Conference*, March 2000.

[3] Barney Pell, et al. "An Autonomous Spacecraft Agent Prototype." *Autonomous Robots* 5(1), March 1998.

***Dr. Erann Gat*** is a senior computer scientist at the Jet Propulsion Laboratory, California Institute of Technology, where he has been working on autonomous control architectures since 1988. He is currently inter-domain architect for the JPL Mission Data System project. He escapes the dangers of everyday life in Los Angeles by pursuing safe hobbies like skiing, scuba diving, and flying small single-engine airplanes in bad weather.