Real Time and Resource Overload Language

Robert S. Hanmer
Lucent Technologies
2000 North Naperville Road
P. O. Box 3033
Naperville, IL 60566-7033
voice: +1 630 979 4786
hanmer@lucent.com

Abstract

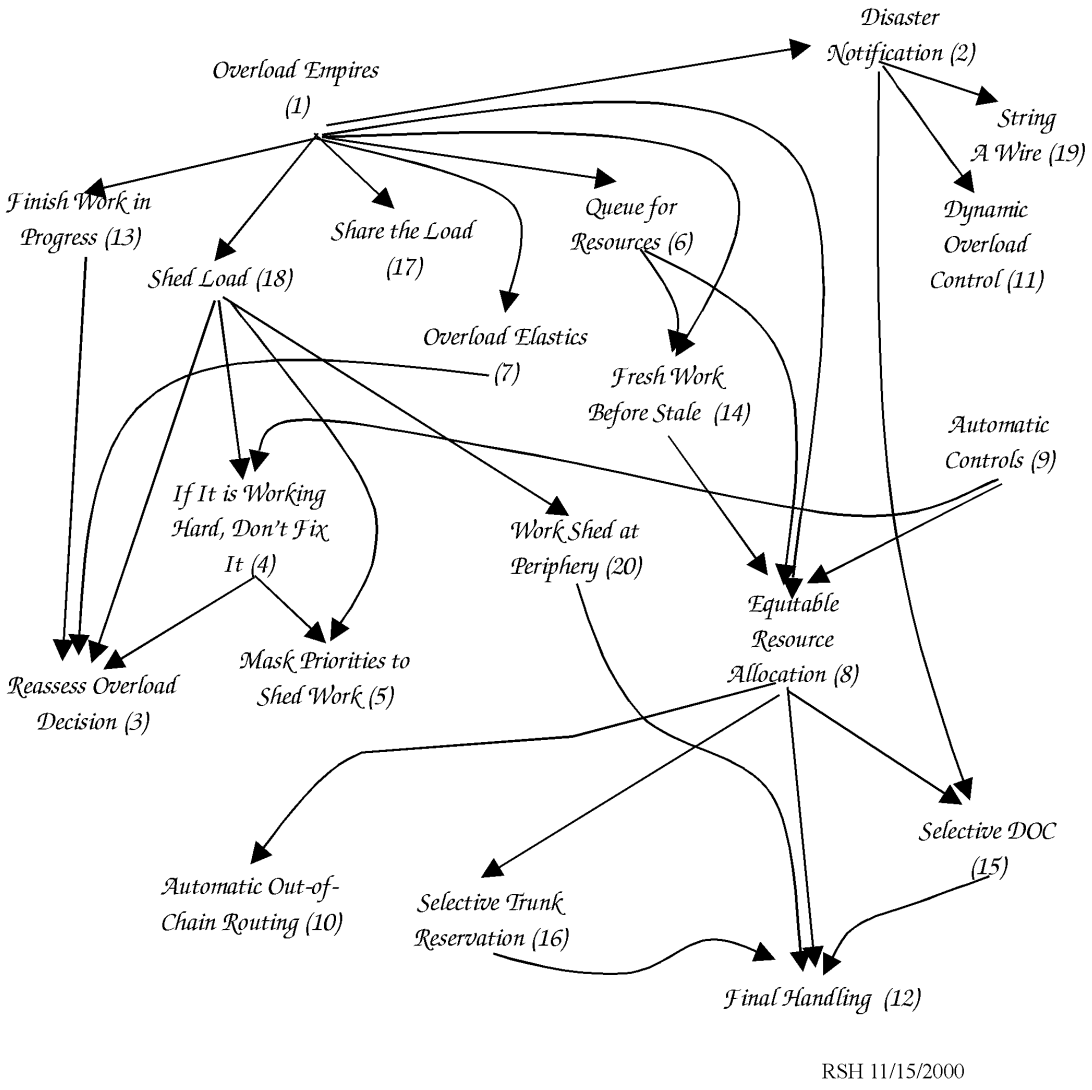
This pattern language merges the Traffic Congestion pattern language from PLoP-99 with the patterns in Gerard Meszaros' "Pattern Language for Improving the Capacity of Reactive Systems" presented in Pattern Languages of Program Design-2 as well as those in the collection of 4ESS™ Switch project from Lucent Technologies.

A system that reacts to externally provided stimuli might receive more stimuli than it can handle at any given time. When this "overload" situation occurs the system must be able to deal with it somehow. Many systems respond by ceasing all work, i.e. locking up or crashing. A well-designed system is able to handle the overload and respond gracefully to the excess stimuli. This system might have somewhat reduced capabilities during the transition, but it will correctly handle much of the work presented to it, and when stimuli levels return to normal, it can resume full functionality with minimal involvement.

These patterns assume that the basics of a fault tolerant architecture exist. For example, the "Fault-Tolerant Telecommunication System Patterns", also in Pattern Languages of Program Design-2 has guided system design so that it will automatically respond to anomalous events.

Language Map

The diagram below shows patterns that enhance the solutions of other patterns, resolve previously unresolved forces in a pattern, or take advantage of an earlier pattern to provide some new system capability. For example, FINISH WORK IN PROGRESS (13) refines the pattern OVERLOAD EMPIRES (1), helping to solve unresolved forces or new problems that OVERLOAD EMPIRES (1) introduced.



Language Context

The patterns within this language are intended to gracefully handle excess stimuli that results in a workload that exceeds the amount of available resources. The resource might be processor CPU time, or they might be a tangible resource such as memory or a tone detector. By gracefully handling whatever traffic arrives at the system, the Quality of Service standards can best be achieved.

When a system is intended to handle errors autonomously, it must first decide if a given system state is due to the manifestation of a fault[1], or an excess of stimuli. This requires that the System Integrity Control (SICO FIRST AND ALWAYS [ACGH+][2]) evaluate to determine global system status. If the Integrity Controller decides that it is indeed a fault situation, fault-handling patterns are to be applied, such as those in the Fault-Tolerant Telecommunications Patterns [ACGH+]. If an overload is present, responses can be grouped into two broad categories: processor and resource. There will be times when both a real-time and resource overload will occur simultaneously. Many techniques apply to only one of these categories, so a mechanism is needed to resolve the question as too which OVERLOAD EMPIRE (1) is overloaded.

To handle a real-time overload situation the system should attempt to FINISH WORK IN PROGRESS (13) to prevent the time spent switching between tasks to exceed the amount of time spent processing tasks. When events arrive that require tangible resources they can be QUEUED FOR THE RESOURCE (6). Recent requests for service should take precedence over those that have been in the queue longer, i.e. do the FRESH WORK BEFORE STALE (14). This is especially important to deal with consumer behaviour.

If the system has idle resources reserved for fault handling, such as spare processors; or if the system has adjunct processors that can help with certain tasks a real-time overload can be handled by SHARING THE LOAD (18) among processors. System architecture sometimes precludes this however. In these cases the system must be designed to SHED LOAD (17) that cannot be handled.

In fault tolerant systems, the system generally has many maintenance tasks executing simultaneously. If the system is performing well yet is overloaded, these tasks can be deferred. In other words, IF IT IS WORKING HARD, DON'T FIX IT (4). Resources must be ALLOCATED EQUITABLY (8). One way of doing this is to use pre-determined allowable PRIORITY MASKS (5) to select which runnable task should be executed.

Work should be shed as close to the edges of the system as possible (WORK SHED AT THE PERIPHERY (20)). The work to bring events into the processors' core, are wasted if the work is to be canceled. When requests for service are canceled, some indicator should be sent to other parties involved through a FINAL HANDLING (12) report.

OVERLOAD ELASTICS (7) can be used to decide the extent of a processor CPU time overload. The system architects must decide what the system should do when attempts to shed work are unsuccessful. One approach to deal with this is found in the pattern OVERLOAD OUT-OF-CONTROL (3): if all other attempts to reduce the level of stimuli are unsuccessful cease processing all new stimuli until the situation improves and real-time becomes available. This might be difficult for some system architects to allow.

Requests for tangible resources should be handled in an EQUITABLE MANNER (8) Requests for these resources are controlled through either protective or expansive AUTOMATIC CONTROLS (9). Expansive controls allow the use of resources that are not normally available for use, such as AUTOMATIC OUT-OF-CHAIN ROUTING (10). Protective controls restrict access to protect the system. Examples of these are SELECTIVE TRUNK RESERVATION (16) and SELECTIVE DYNAMIC OVERLOAD CONTROL (15). Whenever the system cancels and ignores a stimuli FINAL HANDLING (12) should be performed on the stimuli to report status and to aid in diagnosing problems.

---

[1] A *fault* is a deviation from correctness. When a *fault* is encountered in program execution an *error* occurs which is incorrect result. The effect on the system's user is a *failure*.

[2] All the patterns here will be followed by either an internal reference number contained within parenthesis, or a reference to a published paper. Internal reference number one through eight refer to patterns contained within this paper. Internal references greater than eight refer to patterns that are part of this language but are not presented in their entirety here. They are thumbnailed at the end.

1. OVERLOAD EMPIRES

… The situation within the system has been analyzed and the decision has been made that it is not an error caused by faulty hardware or software. Overload situations occur when the system loses the resources necessary to handle its workload efficiently. This might be due to internal problems, such as memory leaks or excessive maintenance work requests which are really faults within the system and should be handled through the fault recovery system (SICO FIRST AND ALWAYS). When external systems send too many requests for service too quickly the system must handle as many as possible and then degrade as smoothly and as little as possible.

❖ ❖ ❖

How should situations of overload be handled?

The resource exhaustion philosophy comes from the days of trunks that needed to have MF[3] trunk receivers quickly connected to the trunk between seizure and digit reception. If MF receivers were in scarce supply, there wouldn't be time to recovery gracefully from the seizure. If we start running out of MF receivers, we tell the far end to reduce what they're sending us, so we can keep up with the traffic. Then, the other system can re-route the traffic. This is a suggestion, not an absolute block: even for switches in overload, another switch may select them as the next link in the routing chain, as the lesser of several evils.

Too many requests for service can be taxing on a system in a number of ways:

- Memory: more memory might be required to store the requests for than the system has available.
- Peripheral equipment: the requests might require the use of tangible peripheral resources that are already in use.
- Processor CPU time: processing the requests might take more time than the system has.

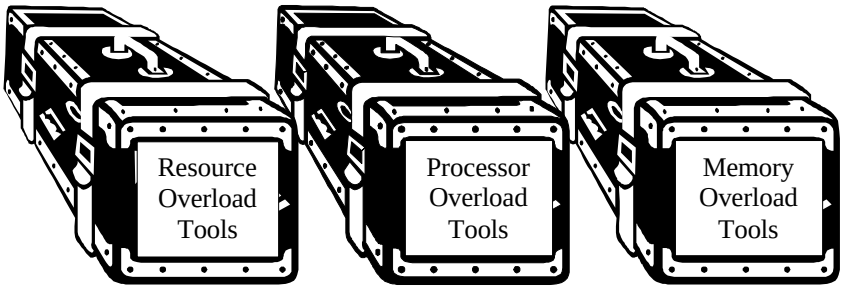
There are a variety of techniques designed to address these resource overloads. Some, such as queuing for memory resources work for some types of requests, but not for others. Some techniques will work for all three. Trying to manage one type of overload with a mechanism designed for another might have devastating results.

Therefore:

---

[3] "MF" or "Multi-Frequency" refers to trunks that use a combination of tones over the actual talking path to exchange call related signals between switching offices. This is a different combination of tones than TouchTone®, which is "Dual Tone Multi-Frequency" or "DTMF".

Administer multiple overload empires, one for managed resources like trunks, lines and peripheral equipment, another for memory, and yet another for processor CPU time. Avoid grouping all of the possibilities together, as they will only rarely work well for overloads in other empires.



❖ ❖ ❖

An effective technique to deal with overloads of the tangible variety is to QUEUE FOR RESOURCES (6). EQUITABLE RESOURCE ALLOCATION (8) discusses a way to divide up the tangible resources such as memory and peripheral equipment.

A technique similar to QUEUING FOR RESOURCES (8) that works for the intangible of CPU cycles is to take on FRESH WORK BEFORE STALE (14). Enabling the system to SHARE LOAD (17) or to SHED LOAD (18) also help with CPU time.

To know whether we are in processor or resource overload there has to be some way of measuring the overload. OVERLOAD ELASTICS (7) discusses metrics that should be used to evaluate overloads.

Consumer/customer behaviour must be considered in deciding how to deal with an excess amount of work. FRESH WORK BEFORE STALE (14) and FINISH WORK IN PROGRESS (13) both discuss a way of dealing with too much work while considering this behaviour.

In a network of peers, strategies can be designed to allow one peer to NOTIFY (2) its neighbors that it is in overload and seek assistance in handling the traffic or in reducing the load from its peers. …

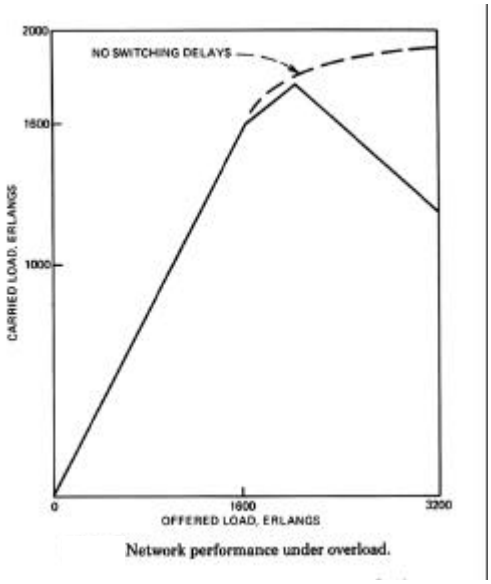
2. DISASTER NOTIFICATION

… The system is in trouble. It might be the result of an excess of requests for service from external sources, or it might be handling errors. In either case the system is dealing with an excessive demand for some kind of resources: CPU time, tangible resources or memory.

❖ ❖ ❖

Overloads happen when too many requests for service arrive too fast. What can a single system do to slow down the influx of requests?

Within a network of systems what happens in one will influence what happens in the others. "Regenerative switching delays, if left uncontrolled, can quickly spread throughout the network, causing the type of decline in carried load shown in [the next figure]." [GHHJ, p. 1170]



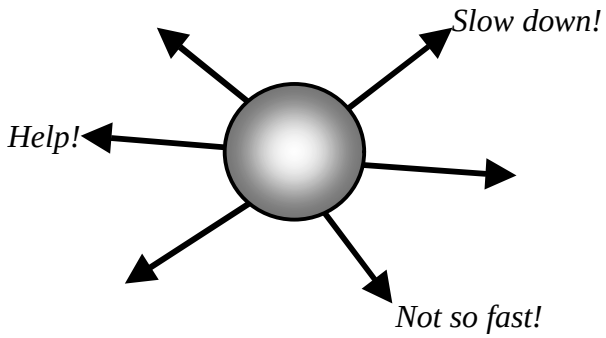
Network performance under overload.

You can resolve much of the internal inefficiency through rigorous testing and good design practices and algorithms. These are things done before the system is placed into service.

External stimuli causes system overloads during execution. By definition, nothing internal can be done when the communicating systems are sending too much traffic to an overloaded system, since the stimuli are "external". But the communicating systems can help if they are informed that they are sending too much information.

Therefore:

Call for help! Institute a method of communication between systems to help throttle the workload at systems in overload. If a system receives such a signal, it should assist by reducing the amount of work being sent to the troubled system.



❖ ❖ ❖

DYNAMIC OVERLOAD CONTROL (11) and SELECTIVE DYNAMIC OVERLOAD CONTROL (15) are examples of such mechanisms for different types of overload responses.

STRING A WIRE (19) from the Telecommunications Input Output Language [HS] describes how these signals can be sent. By using a fixed, permanent connection few of the overloaded system resources will be used to send the signal. …

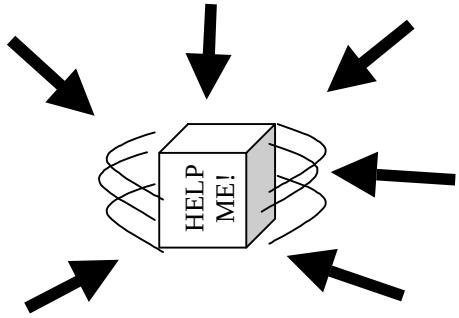
3. REASSESS OVERLOAD DECISION[4]

… The system is attempting to FINISH WORK IN PROGRESS (13) as well as to SHED LOAD (18).

❖ ❖ ❖

What should the system do when the usual load reduction techniques are not working to diminish the workload?

What happens if load keeps increasing in spite of all attempts to slow the system down?



The system is well engineered so that work shedding keeps the system from going into deep saturation. The mechanisms instituted to SHED LOAD (18) are working, yet the influx of new requests or the compounding of internal inefficiencies are not producing the desired reduction in workload. These mechanisms create a negative feedback loop that should keep load from getting out of hand.

Something's wrong if we haven't had any new requests for service in a long time. The system is designed to perform some work, such as to process telephone calls. If that is skipped for too long a period of time, it doesn't make any money for its owner.

A major goal of the overload handling mechanisms is to preserve system sanity[5] so that when the overload period is ended the system can handle the routine level of traffic.

---

[4] Strategy alluded to in [GHHJ, p. 1177]

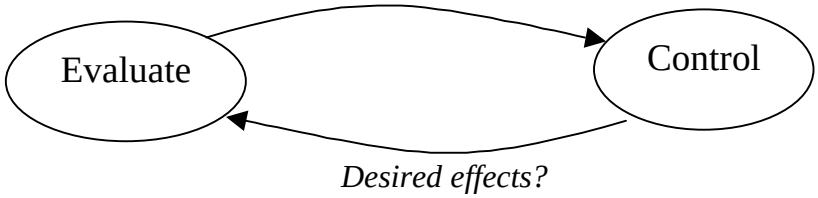
[5] Sanity as it is used here refers to the system executing as designed with a clear task or set of tasks in control of the Program Counter in some manner intended by the system's developers.

If feedback isn't enough to bring the system out of saturation, then the overload may be coming from a source other than traffic.

For example, when there's congestion (like MF trunks) overload will make a note that this facility type is congested, and doesn't do anything. It checks again some time later: if the congestion is still there, it applies an overload control or might trigger fault handling.

Therefore:

Provide the ability for the system to reexamine its decision that this is an overload instead of an error. This might mean that the system decides that it is not an overload but really an error.



❖ ❖ ❖

Unless reexamination is possible the system can get further and further into trouble by following the wrong path. This is related to SICO FIRST AND ALWAYS [ACGH+] and could use the same Integrity Controller to make the decisions.

4. IF IT IS WORKING HARD, DON'T FIX IT[6]

… More work is arriving than the system can handle. The system is SHEDDING LOAD (18) and thus passing up revenue opportunities because it must be able to actually complete some work in order to realize the revenue.

❖ ❖ ❖

What work should be shed?

There aren't enough CPU resources both to handle the capacity and to continue the overhead work. This overhead includes the auditing and maintenance functions that keep the system fault tolerant. It might be skipping some of its main application work already due to congestion. The choice is to reduce even more the revenue producing work or to restrict some of the activities that guarantee the system's fault tolerance.

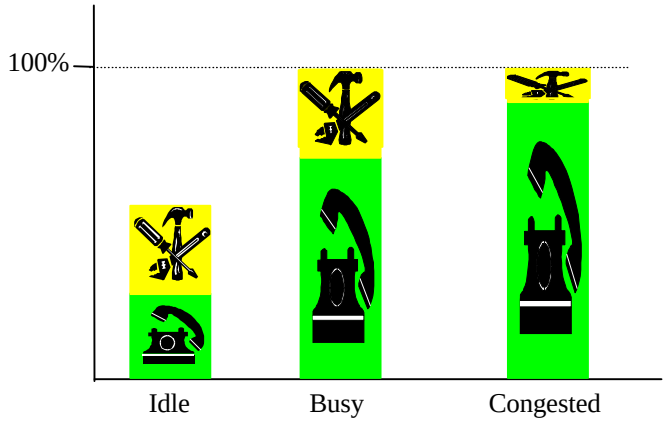
The system has very stringent availability requirements, which is why a system of audits, defensive checks and integrity monitors in place. These parts of the system ensure that the system is working at its peak efficiency and detect errors to contain and correct them.

But if we're in traffic induced overload, the trunks must be working and software must be working, since we're doing work--so let's defer the stuff that comes to play when we're *not* working properly. These deferrable items do work that isn't critical to the primary application. If it works, don't check if it works--release the time so that we can concentrate on the primary money making aspects of operation.

Therefore:

Defer maintenance work. Use the system's task scheduler to implement this strategy. If the system is tending toward overload, chances are that the trunks and software are working--otherwise, where would all that work be coming from?

[6] Strategy alluded to in [GHHJ, p. 1177]

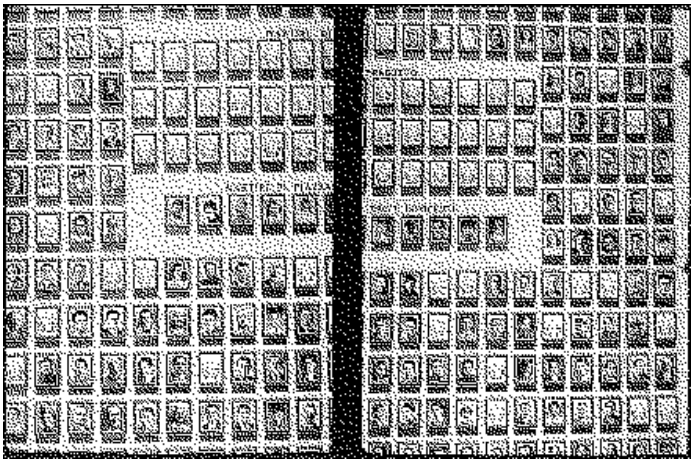


❖ ❖ ❖

There's an outside chance that the system just *seems* like it's in overload, though it may be really reacting to errors. In that case, employ REASSESS OVERLOAD DECISION (3). REASSESS OVERLOAD DECISION (3) also addresses when this strategy is not working sufficiently and the system isn't recovering from the overload. MASK PRIORITIES TO SHED WORK (5) discusses one way that this pattern can be implemented.

Everything that the system does is important to someone. But not everything is directly related to the primary purpose of the system. Tasks should not deferred forever. MASK PRIORITIES TO SHED WORK (5) provides an equitable way to do this. …

5. MASK PRIORITIES TO SHED WORK



… You want to SHED LOAD (18) and in particular you are implementing IF IT IS WORKING HARD DON'T FIX IT (4).

❖ ❖ ❖

How do you spread out the workload under overload without skewing priorities?

There are many ways to select certain tasks to defer temporarily. Some involve development time decisions of what is more important. Some involve execution time decisions, as in IF IT IS WORKING HARD DON'T FIX IT (4). The best way is something that is fair.

All the work that the system performs is important; nothing should be totally eliminated during overload. Requirements upon execution frequency may be stretched, but eventually all tasks need to be scheduled.

If we were to alternate tasks that normally are all executed, and execute one half this time and the other half on the next, every task would be executed eventually. The time period between successive task executions would be increased, but during periods of overload everything is running more slowly, so this is acceptable. The time between subsequent iterations with this alternation might be less than if nothing was done.

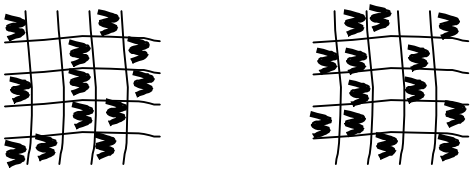
One way of implementing this is through the use of bit masks. Every task should have its allowable bit set in at least one of the masks. Tasks that are runnable and allowable will be executed. Some tasks that are "more" important might have bits set in several masks so that they get entered more frequently.

Ready Tasks:	`10110011`
Run now:	`10100110`
Run next time:	`01011011`

Sometimes there are interactions between multiple tasks that will dictate that certain of them must be executed together (or without certain others intervening).

Therefore:

Use bit masks to overlay the runnable task words. Every scheduling loop, overload toggles between several of these "allowable" masks. By alternating allowable masks, and making sure that every task appears in the masks, every task will eventually get scheduled.



❖ ❖ ❖

When several tasks interact strongly and the mask mechanism might not guarantee correctness the general scheduler and its mask might not be the appropriate scheduling technique.

Good engineering judgement is required to determine how tasks should be sorted onto the different masks.

6. QUEUE FOR RESOURCES[7]

… The system is overloaded, and not in the midst of failure processing. Too many requests for tangible services such as memory or peripheral equipment (such as MF receivers) are being received. (OVERLOAD EMPIRES (1)).

❖ ❖ ❖

What should be done with requests for tangible resources that cannot be handled at the moment?

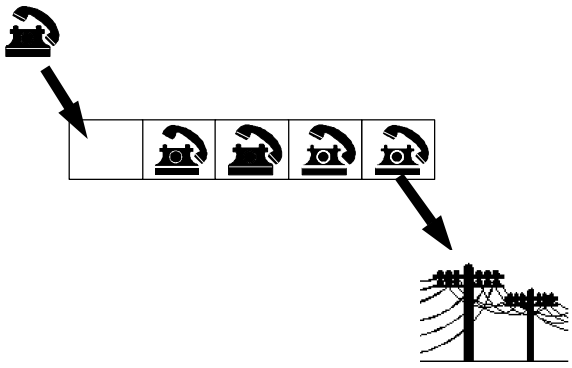
The system is receiving too many stimuli for the moment, but in general can handle the excess requests for service.

You could throw up your hands and reject all requests that can't be handled as they appear. This supports the pattern FRESH WORK BEFORE STALE (14). But it results in work that if deferred for only a short period might be handled.

If you can store the work in a queue for later processing then the work might complete eventually. The risks of this are that the queue might get longer than can be effectively managed. There is also the risk that the work won't need to be done when the task is finally ready.

Therefore:

Store requests for service that cannot be handled immediately in a queue. Give the queue a finite length to improve the likelihood that the request is still necessary when it reaches the head of the line.



[7] Reference: [WWF]

❖ ❖ ❖

The queue should use a LIFO strategy (as in FRESH WORK BEFORE STALE (4)*)* to govern insertion and removal from the queue. This will help people think that they are receiving good service. Allocation of resources under the guidance of EQUITABLE ALLOCATION (8) should recognize both the requests that have been queued and those that are fresh and have never been queued. …

7. OVERLOAD ELASTICS[8]

… The problem appears to be one of processor CPU time overload (OVERLOAD EMPIRES (1)). This is an overload of an intangible resource.

❖ ❖ ❖

How should we judge the severity of the too many requests for resources?

Artificial indicators can be created to measure the severity of the overload. This introduces additional overhead that will be most needed just when the system has the least resources available.

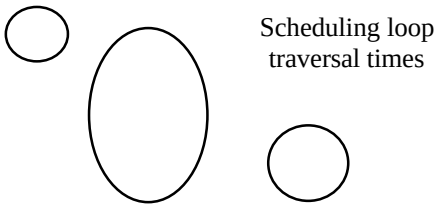
Or already existing indicators can be used. Some indicator such as per cent CPU idle time can be used. This does not increase overhead, since the computation already done.

The CPU idle time is a metric that system designers include as a measure of workload and its variability.

In some scheduling regimes, such as round robin, there is no idle time by definition. Such is the case in many real time systems. Generally in these cases some sort of existing measurement, similar to idle time, is used to allow the system owners to gauge its performance. In these systems the length of time spent traversing the loop appears quite elastic.

Therefore:

Use an indicator already tied to the resource as an indicator of the system's sanity and overload condition.



❖ ❖ ❖

It is important to periodically REASSESS THE OVERLOAD DECISION (3) by checking the overload indicators.

[8] Strategy alluded to in [CCRSS, P. 1116]

8. EQUITABLE RESOURCE ALLOCATION

… You are trying to handle FRESH WORK BEFORE STALE (14) and yet you have many requests QUEUED FOR RESOURCES (6). There are distinct types of resources that need to be allocated to requests. The system is prepared and capable of instituting AUTOMATIC CONTROLS (9).

❖ ❖ ❖

How should requests for scarce resources be handled?

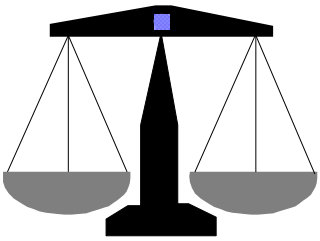
You could strictly follow FRESH WORK BEFORE STALE (14) and only give the newest request service, even if they are from predominantly one type/class/area. Customers might be paying for a premium service however and will not appreciate being lumped into the resource allocation pool with the bargain-rate customers.

There might be a specific resource that is especially overloaded. If requests are allocated based only upon their newness, i.e. position in the queue, then they might end up blocking on this resource anyway.

Another strategy would be to look at all the requests for service, both fresh and queued and allocate resources equitably to all of them. While this requires additional bookkeeping be done, work can be directed around extremely specific resource overloads. This helps ensure the greatest common good by providing service to as many requests as possible.

Therefore:

Pool all similar requests and allocate resources to the pools based upon their availability and priority. This allows all types of work to be accomplished even if concentrated overloads from a certain category of type/class/area exist.



❖ ❖ ❖

FINAL HANDLING (12) is required for those requests for service that are abnormally terminated.

When inter-office trunks are the resources being allocated, there are several patterns that can help. When the incoming traffic is within groups that have few idle members, SELECTIVE TRUNK RESERVATION (16) can be helpful. SELECTIVE DYNAMIC OVERLOAD CONTROL (SELECTIVE DOC) (15) is useful to restrict traffic of certain types through DISASTER NOTIFICATION (2) of congestion so that they will restrict traffic flow. If the flow of traffic is extremely mismatched then the expansive control of AUTOMATIC OUT-OF-CHAIN ROUTING (10) can help. …

Previously Published Patterns

Internal Reference Number	Pattern	Source	Intent
9	**AUTOMATIC CONTROLS**	[HW]	When conditions dictate, the switch should automatically institute changes to normal behaviour to respond to conditions.
10	**AUTOMATIC OUT-OF-CHAIN ROUTING**	[HW]	During overloaded periods, allow new routes within a hierarchical network.
11	**DYNAMIC OVERLOAD CONTROL**	[HW]	Provide mechanism to tell far-ends to slow down.
12	**FINAL HANDLING**	[HW]	Gracefully tear down a call that cannot complete.
13	**FINISH WORK IN PROGRESS**	[MESZ]	Give priority to work that continues already in progress work.
14	**FRESH WORK BEFORE STALE**	[MESZ]	Give better service to recent requests.
15	**SELECTIVE DYNAMIC OVERLOAD CONTROL**	[HW]	Measure the length of the infinite loop to decide on Overloads.
16	**SELECTIVE TRUNK RESERVATION**	[HW]	Deny incoming traffic on TSGs (trunk subgroups) that have few idle trunks during periods of congestion.
17	**SHARE THE LOAD**	[MESZ]	Move some processing to another processor.
18	**SHED LOAD**	[MESZ]	Throw away some requests for service to offer better service to other requests.
19	**STRING A WIRE**	[HS]	Provide a system-to-system emergency information channel.
20	**WORK SHED AT PERIPHERY**	[MESZ]	Shed work at minimal cost where it first enters the system.

Acknowledgments

Mike Adams was a co-author on previous versions of DYNAMIC OVERLOAD CONTROL, EQUITABLE ALLOCATION, EVALUATE OVERLOAD GLOBALLY, OVERLOAD EMPIRES, IF IT'S WORKING HARD, DON'T FIX IT and DISASTER NOTIFICATION.

Ward Cunningham was PLoP 200 shepherd for this language.

Karen Hanmer graciously scanned several images to accompany these patterns.

The photo with Mask Priorities to Shed Work is *Italian Dyptic 5* copyright 1997 Steve Harp, used with permission of the artist.

Thanks to my PloP2K Writers' Workshop group for their valuable comments. Bill Opdyke, Carlos O'Ryan, Brian Foote, Rossana Andrade, Todd Coram, Brian Marick, Juha Pärssinen and Terunobu Fujino were members of this group, entitled "Network of Learning"

References

[ACGH+] Adams, M., J. Coplien, R. Gamoke, R. Hanmer, F. Keeve and K. Nicodemus. 1996. "Fault-Tolerant Telecommunication System Patterns" in **Pattern Languages of Program Design - 2,** edited by J. M. Vlissides, J. O. Coplien and N. L. Kerth. Reading, MA: Addison-Wesley Publishing Co.

[CCRSS] Cieslak, T., L. Croxall, J. Roberts. M. Saad, and J. Scanlon, 1977. "No 4 ESS: Software Organization and Basic Call handling." **Bell System Technical Journal**, vol 56, no. 7, Sept, 1977: 1113-1138.

[GHHJ]: Green, T. V., D. G. Haenschke, B. H. Hornbach and C. E. Johnson. 1977. "No 4 ESS: Network Management and Traffic Administration." **Bell System Technical Journal**, vol. 56, no. 7, Sept, 1977: 1169-1202.

[HS] Hanmer, R., and G. Stymfal, 1999. "An Input and Output Pattern language: Lessons from Telecommunications" in **Pattern Languages of Program Design - 4**, edited by N. Harrison, B. Foote and H. Rohnert. Reading, MA: Addison-Wesley Publishing Co.

[HW] Hanmer, R., and M. Wu, 1999. "Traffic Congestion Patterns". Presented and workshoped at PLoP-99 conference. http://st-www.cs.uiuc.edu/~plop/plop99/proceedings/

[Mesz] Meszaros, G. 1996. “A Pattern Language for Improving the Capacity of Reactive Systems” in **Pattern Languages of Program Design - 2,** edited by J. M. Vlissides, J. O. Coplien and N. L. Kerth. Reading, MA: Addison-Wesley Publishing Co.

[WWF] Wake, W., B. Wake, and E. Fox. "Improving Responsiveness in Interactive Applications Using Queues" in **Pattern Languages of Program Design - 2**, edited by J. M. Vlissides, J. O. Coplien and N. L. Kerth. Reading, MA: Addison-Wesley Publishing Co.