Chapter 2

Performance Antipatterns

In the last two decades, the concept of performance antipattern has been used for "codifying" the knowledge and experience of software analysts [151, 152, 155, 153]. *Performance antipatterns* are well-known bad design practices that lead to software products suffering by poor performance; their definitions include refactoring actions that can be carried out to remove them. Starting from textual specifications [155], a set logic-based definitions of performance antipatterns have been provided in [51].

We present in this chapter both the original textual definitions and the logic-based one that we consider as starting point for this thesis' work. First, an overview of performance antipatterns and their main characteristics is provided. Then, six antipatterns (i.e., Blob, Concurrent Processing Systems, Pipe and Filter Architectures, Extensive Processing, Empty Semi-Trucks, and Traffic Jam) are deeply described, and they will be referred in the rest of this thesis for providing proof-of-concepts.

2.1 Textual Specifications

Antipattern occurrences can be detected in a software architectural model and refactoring actions can be taken for solving each antipattern. Smith and Williams have been in principle the most prolific researchers that, basing on their practical experience, have defined fourteen notation- and domain-independent performance antipatterns [151, 152, 155, 153]. Two antipatterns have been removed from the original list for the following reasons: the *Falling Dominoes* antipattern refers not only to performance problems, it includes also reliability and fault tolerance issues, therefore it is out of this thesis scope; the *Unnecessary Processing* antipattern deals with the semantics of the processing by judging the importance of the application code, therefore it works at an abstraction level typically not observed in software models.

Table 2.1 lists the remaining performance antipatterns that represent our main reference in this domain. Each row represents an antipattern as characterized by three attributes: *antipattern* name, *problem* description, and *solution* description.

The list of performance antipatterns has been here enriched with an additional attribute. As shown in the leftmost part of Table 2.1, we have partitioned antipatterns

Table 2.1: Performance Antipatterns: problem and solution textual descriptions.

	Antipattern		Problem	Solution
Single-value	Blob (god class/component)		Occurs when a single class or component either 1) performs all of the work of an application or 2) holds all of the application's data. Either manifestation results in excessive message traffic that can degrade performance.	Refactor the design to distribute intelligence uniformly over the application's top-level classes, and to keep related data and behavior together.
	Unbalanced Processing	Concurrent Processing Systems	Occurs when processing cannot make use of available processors.	Restructure software or change scheduling algorithms to enable concurrent execution.
		"Pipe and Filter" Architectures	Occurs when the slowest filter in a "pipe and filter" architecture causes the system to have unacceptable throughput.	Break large filters into more stages and combine very small ones to reduce overhead.
		Extensive Processing	Occurs when extensive processing in general impedes overall response time.	Move extensive processing so that it does not impede high traffic or more important work.
	Circuitous Treasure Hunt		Occurs when an object must look in several places to find the information that it needs. A large amount of processing per look worsens performance.	Refactor the design to provide alternative access paths that do not require a Circuitous Treasure Hunt (or to reduce the cost of each look).
	Empty Semi Trucks		Occurs when an excessive number of requests is required to perform a task. It may be due to inefficient use of available bandwidth, an inefficient interface, or both.	The Session Facade design pattern provides more efficient interfaces and a better use of available bandwidth [159].
	Tower of Babel		Occurs when processes excessively convert, parse, and translate internal data into a common exchange format such as XML.	The Fast Path performance pattern identifies paths that should be streamlined. Minimize the conversion, parsing, and translation on those paths by using the Coupling performance pattern to match the data format to the usage patterns.
	One-Lane Bridge		Occurs at a point of execution where a few processes may continue to execute concurrently (e.g., when accessing a database). Other processes are delayed while they wait for their turn.	To alleviate the congestion, use the Shared Resources Principle to minimize conflicts.
	Excessive Dynamic Allocation		Occurs when an application unnecessarily creates and destroys large numbers of objects during its execution. The overhead required to creations and destructions has a negative impact on performance.	1) Recycle objects (via an object pool) rather than creating new ones each time they are needed. 2) Use the Flyweight pattern to eliminate the need to create new objects.
Multiple-values	Traffic Jam		Occurs when one problem causes a backlog of jobs that produces wide variability in response time which persists long after the problem has disappeared.	Begin by eliminating the original cause of the backlog. If this is not possible, provide sufficient processing power to handle the worst-case load.
	The Ramp		Occurs when processing time increases as the system is used.	Select algorithms or data structures based on maximum size or that adapt to the size.
	More is Less		Occurs when a system spends more time thrashing than accomplishing real work because there are too many processes relative to available resources.	Quantify the thresholds where thrashing occurs (using models or measurements) and determine if the architecture can meet its performance goals while staying below the thresholds.

in two different categories: *Single-* and *Multiple-value* [52]. The former category collects antipatterns detectable by single values of performance indices (such as mean, max or min values). The latter category collects those antipatterns requiring the trend (or evolution) of performance indices over time (i.e. multiple values) to capture the performance problems induced in the software system. Mean, maximum or minimum values are not sufficient for characterizing multiple-value antipatterns unless these values are referred to several observation time frames. Due to this aspect, the performance indices needed to detect such antipatterns must be obtained via system simulation or monitoring.

2.2 Logic-based Specifications

A formal representation of performance antipatterns have been formally defined as logical predicates [51]. Such predicates define conditions on specific software model elements (e.g. number of interactions among software resources, services response time) under which antipatterns occur. This formalization of antipatterns reflects an interpretation of their informal textual definitions.

Performance antipatterns are very complex (as compared to other software patterns) because they are founded on different characteristics of software systems, spanning from static through behavioral to deployment. They additionally include parameters related to design features (e.g., *many* usage dependencies, *excessive* message traffic) and performance indices (e.g., *high*, *low* utilization). Hence, thresholds must be introduced on them.

Parameters and their thresholds have been defined by introducing, respectively, *supporting functions* that extract the information of interest from the software model, and *thresholds* to which extracted values need to be compared. In the logic-specifications of performance antipatterns, the former are denoted as $F_{funcName}$, whereas the latter are denoted as $Th_{thresholdName}$. For example, $F_{numMsgs}$ and $F_{maxHwUtil}$ are defined for the Blob antipattern (see Section 2.3): the former counts the number of messages sent from a software entity to another one in a service; the latter provides the maximum utilization among the hardware devices of a certain type (e.g., cpu, disk) hosted by a processing node.

Thresholds must be instantiated to concrete numerical values, e.g., hardware resources whose utilization is higher than 0.8 can be considered as problematic. The calibration of these values is a critical point of the whole SPE process (see Figure 1.3), because they introduce uncertainty and must be suitably tuned. Some sources can be used to perform this task such as: (i) the system requirements; (ii) the domain expert's knowledge; (iii) the estimation of the system under analysis. Functions and thresholds introduced for representing reference antipatterns as logical predicates are reported in Section 2.3. The multiplicity and the estimation accuracy of thresholds is a key point of this thesis, and it is deeply investigated in Chapter 6.

As stated in Section 1.5, we refer to a multi-view annotated software model, composed by *Static*, *Dynamic* and *Deployment Views*, where each view is properly annotated with parameters involved in the analysis process [25]. The benefit of introducing such views is that performance antipattern specifications can be partitioned on their basis: the predicate expressing a performance antipattern is in fact the conjunction of sub-predicates, each referring to a different view. However, to specify an antipattern it might not be necessary information coming from all views, because certain antipatterns involve only elements of some views (e.g. Concurrent Processing System).

2.3 Reference Performance Antipatterns

In this section, we present examples of the Blob, Concurrent Processing Systems (CPS), Pipe and Filter Architectures (P&F), Extensive Processing (EP), Empty Semi-Trucks (EST), and Traffic Jam (TJ) performance antipatterns [151, 152, 155, 153], which are the ones we will use throughout the rest of this thesis to provide proof-of-concepts when needed.

2.3.1 The Blob

Problem *- Occurs when a single class or component either 1) performs all of the work of an application or 2) holds all of the application's data. Either manifestation results in excessive message traffic that can degrade performance.*

The Blob antipattern may occur in two different cases.

In the first case, a single class or component contains most part of the application logics, while all the other classes or components are used as data containers that offer only accessing functions, i.e. typically *get()* and *set()* methods. The typical behavior of such kind of class or component, called *Blob-controller* in the following, is to get data from other classes or components, perform a computation and then update data on the other classes or components.

In the second case, a single class or component is used to store most part of the data of the application without performing any logic. A large set of other classes or components perform all the computation by getting data from such kind of class or component, called *Blob-dataContainer* in the following, through its *get()* methods and by updating data through its *set()* methods.

Both forms of the blob result from a poorly distributed system intelligence, i.e., a poor design that splits data from the relative processing logic. It might occurs in legacy systems, often composed by a centralized computing entity or by a unique

data container, that are upgraded to object oriented ones without a proper re-engineering design analysis. The performance impacts of both cases are mainly due to the consequent excessive message passing among the blob software entity and the other classes or components. The performance loss is clearly heavier on distributed systems, where the time needed to pass data between remote software entities is significant with respect to the computational time.

Solution - *Refactor the design to distribute intelligence uniformly over the application's top-level classes, and to keep related data and behavior together.*

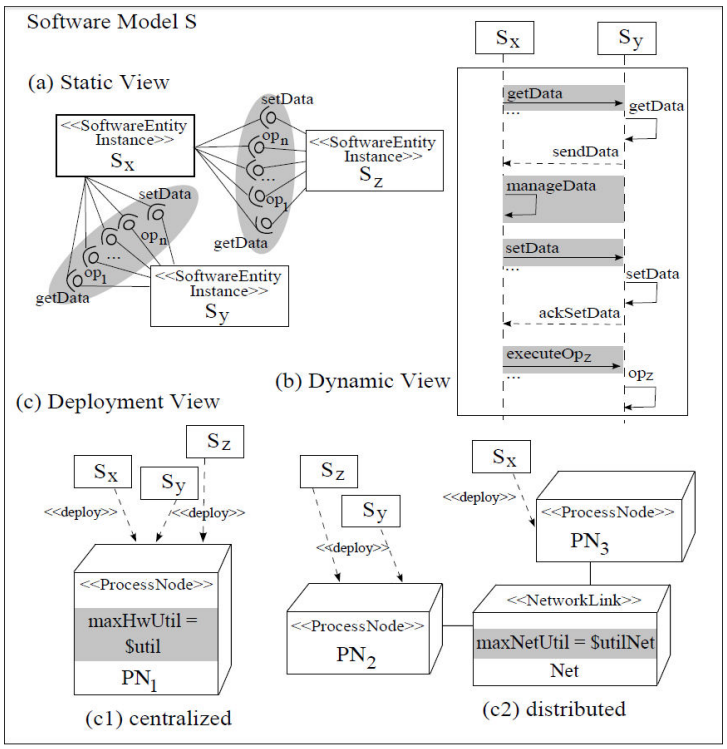
The solution to the blob antipattern is to refactor the design, because it is important to keep related data and behavior together. A software entity should keep most of the data that it needs to make a decision. The performance gain for the refactored solution will be $T_s = M_s \times O$, where T_s is the processing time saved, M_s is the number of messages saved and O is the overhead per message. The amount of overhead for a message will depend on the type of call, for example a local call will have less overhead than a remote procedure call.

Figures 2.1 and 2.2 provide a graphical representation of the *Blob* antipattern in its two forms, i.e., *Blob-controller* and *Blob-dataContainer* respectively.

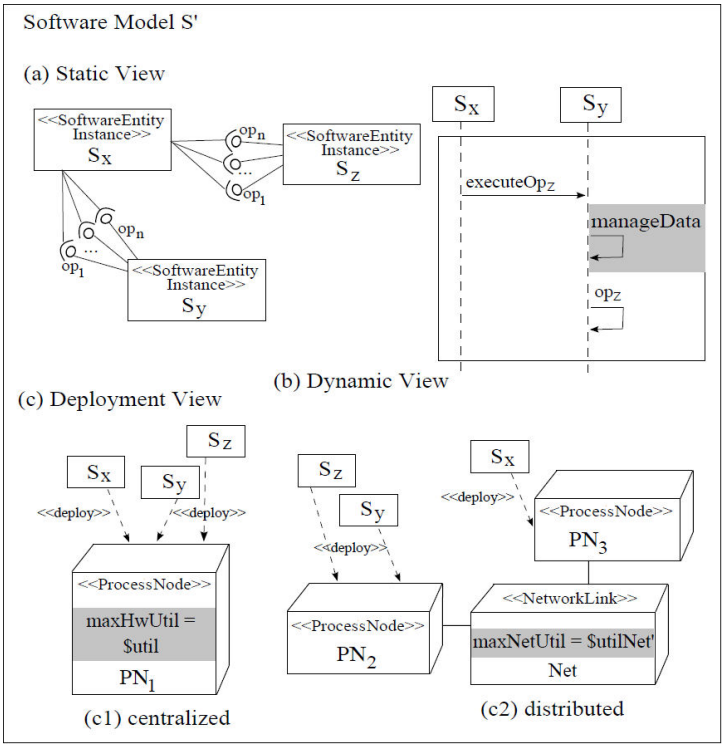
The upper side of Figures 2.1 and 2.2 describes the properties of a *Software Model S* with a *Blob problem*: (a) *Static View*, a complex software entity instance, i.e. S_x, is connected to other software instances, e.g. S_y and S_z, through *many* dependencies; (b) *Dynamic View*, the software instance S_x generates (see Figure 2.1) or receives (see Figure 2.2) *excessive* message traffic to elaborate data managed by other software instances such as S_y; (c) *Deployment View*, it includes two sub-cases: (c1) the centralized case, i.e. if the communicating software instances are deployed on the same processing node then a shared resource will show *high* utilization value, i.e. $\$util$; (c2) the distributed case, i.e. if the communicating software instances are deployed on different nodes then the network link will be a critical resource with a *high* utilization value, i.e. $\$utilNet$ [1]. The occurrence of such properties leads to assess that the software resource S_x originates an instance of the Blob antipattern.

The lower side of Figures 2.1 and 2.2 contains the design changes that can be applied according to the *Blob solution*. The following refactoring actions are represented: (a) the number of dependencies between the software instance S_x and the surrounding ones, like S_y and S_z, must be decreased by delegating some functionalities to other instances; (b) the number of messages sent (see Figure 2.1) or received (see Figure 2.2) by S_x must be decreased by removing the management of data belonging to other software instances. As consequences of previous actions: (c1) if the communicating software instances were deployed on the same hardware resource then the latter will not be a critical resource anymore, i.e. $\$util' \ll \$util$; (c2) if

[1] The characterization of antipattern parameters related to system characteristics (e.g. *many* usage dependencies, *excessive* message traffic) or to performance results (e.g. *high*, *low* utilization) is based on thresholds values (see more details in Chapter 6).

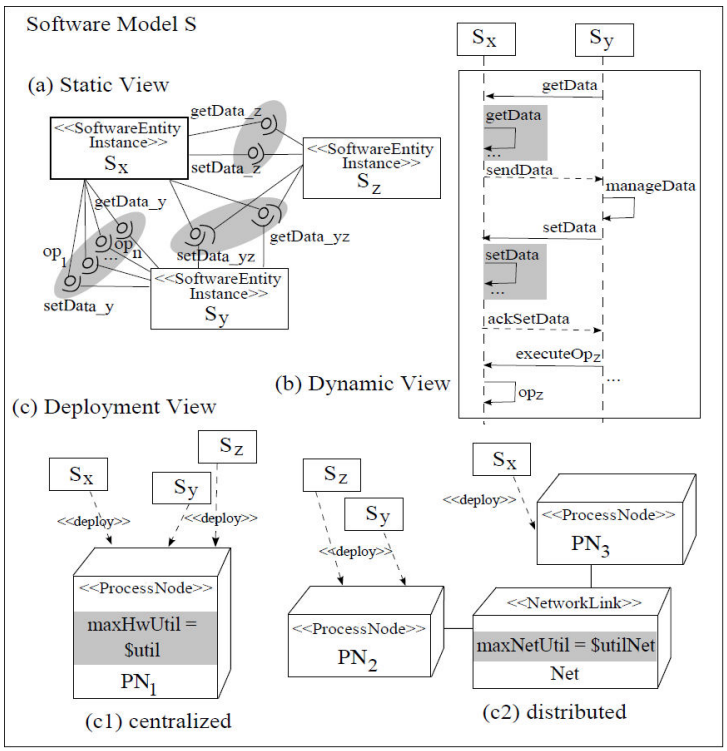


(a) Blob-controller problem.

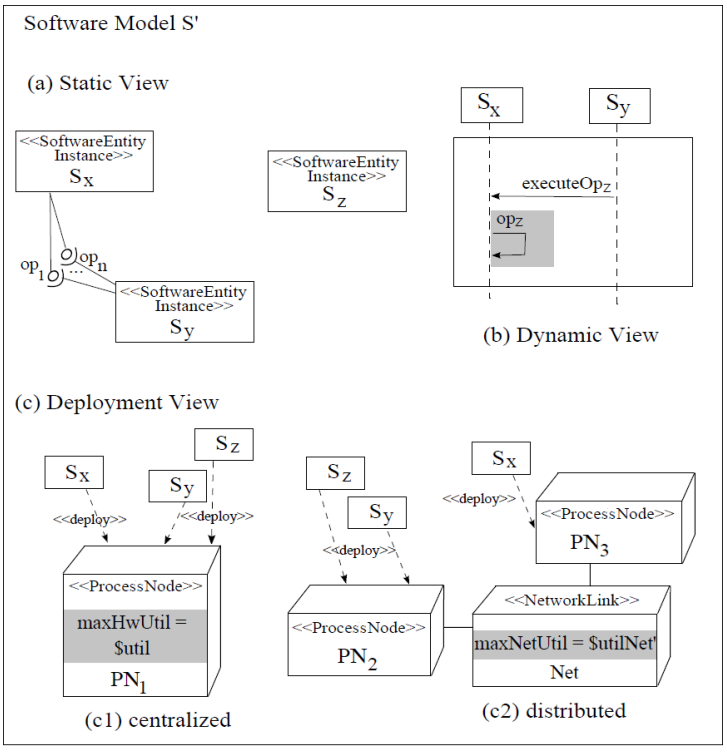


(b) Blob-controller solution.

Figure 2.1: A graphical representation of the Blob-controller performance antipattern.



(a) Blob-dataContainer problem.



(b) Blob-dataContainer solution.

Figure 2.2: A graphical representation of the Blob-dataContainer performance antipattern.

the communicating software instances are deployed on different hardware resources then the network will not be a critical resource anymore, i.e. $\$utilNet' \ll \$utilNet$.

The logic formula of the Blob antipattern has been defined in [51] and reported in Equation (2.1), where swℍ and 𝕊 represent the set of all software components and services, respectively.

$$\begin{aligned} &\exists swE_x, swE_y \in \text{sw}\mathbb{E}, S \in \mathbb{S} \mid \\ &(F_{numClientConnects}(swE_x) \geq Th_{maxConnects} \\ \vee\ &F_{numSupplierConnects}(swE_x) \geq Th_{maxConnects}) \\ \wedge\ &(F_{numMsgs}(swE_x, swE_y, S) \geq Th_{maxMsgs} \\ \vee\ &F_{numMsgs}(swE_y, swE_x, S) \geq Th_{maxMsgs}) \\ \wedge\ &(F_{maxHwUtil}(P_{xy}, all) \geq Th_{maxHwUtil} \\ \vee\ &F_{maxNetUtil}(P_{swE_x}, P_{swE_y}) \geq Th_{maxNetUtil}) \end{aligned} \tag{2.1}$$

Table 2.2 reports the functions involved in the Blob specification. The first column of the Table shows the function signatures and the second column provides their descriptions.

Table 2.2: Functions specification for the Blob performance antipattern.

Function	Description
int $F_{numClientConnects}$ (SoftwareEntityInstance swE_x)	It counts the multiplicity of the relationships where swE_x assumes the client role.
int $F_{numSupplierConnects}$ (SoftwareEntityInstance swE_x)	It counts the multiplicity of the relationships where swE_x assumes the supplier role.
int $F_{numMsgs}$ (SoftwareEntityInstance swE_x, SoftwareEntityInstance swE_y, Service S)	It counts the number of messages sent from swE_x to swE_y in a service S.
float $F_{maxHwUtil}$ (ProcessNode pn_x, type T)	It provides the maximum utilization among the hardware devices of a certain type T = {cpu, disk, all} hosted by the processing node pn_x.
float $F_{maxNetUtil}$ (ProcessNode pn_x, ProcessNode pn_y)	It provides the maximum utilization among the network links joining the processing nodes pn_x and pn_y.

Table 2.3 reports the thresholds involved in the Blob specification [51]: two thresholds ($Th_{maxConnects}$, $Th_{maxMsgs}$) refer to design features, whereas the other ones ($Th_{maxHwUtil}$, $Th_{maxNetUtil}$) are related to performance indices.

Table 2.3: Thresholds specification for the Blob performance antipattern.

	Threshold	Description
Design	$Th_{maxConnects}$	Maximum bound for the number of connections in which a component is involved.
	$Th_{maxMsgs}$	Maximum bound for the number of messages sent by a component in a service.
Performance	$Th_{maxHwUtil}$	Maximum bound for the hardware device utilization.
	$Th_{maxNetUtil}$	Maximum bound for the network link utilization.

2.3.2 Concurrent Processing Systems (CPS)

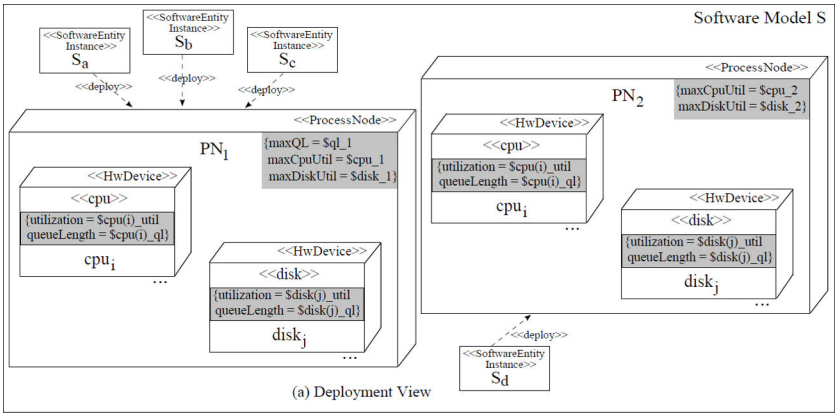
Problem *- Occurs when processing cannot make use of available processors.*

The Concurrent Processing Systems antipattern represents a manifestation of the Unbalanced Processing antipattern [153]. It occurs when processes cannot make effective use of available processors either because of 1) a non-balanced assignment of tasks to processors or because of 2) single-threaded code. In the following we only consider the case 1), since the application code is an abstraction level typically not included in architectural models. As a consequence of the considered case, some hardware nodes are *over-utilized* and some others are *under-utilized*. CPS occurrences are denoted with ($hwNode_1$, $hwNode_2$), where $hwNode_1$ is the over-utilized hardware node and $hwNode_2$ is the under-utilized one.

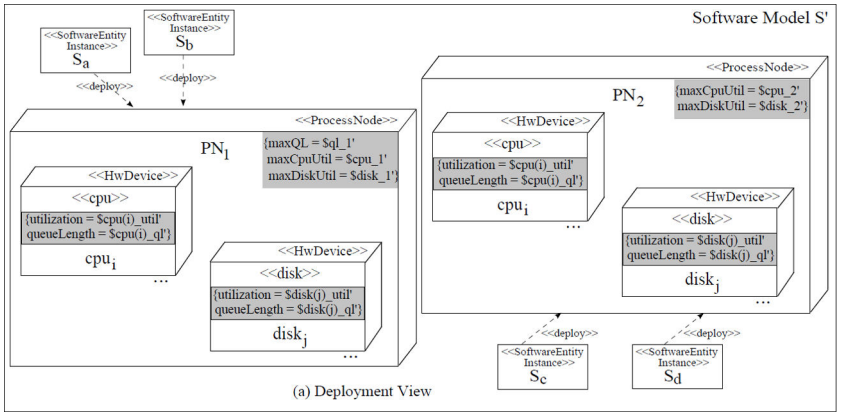
Solution - *Restructure software or change scheduling algorithms to enable concurrent execution.*

If a routing algorithm is based on static properties that result in more work going to one queue than others, than the solution is to use a dynamic algorithm that routes work to queues based on the work requirements and the system congestion.

Figure 2.3 and provides a graphical representation of the CPS antipattern.



(a) Concurrent Processing Systems problem.



(b) Concurrent Processing Systems solution.

Figure 2.3: A graphical representation of the Concurrent Processing Systems performance antipattern.

The upper side of Figure 2.3 describes the system properties of a *Software Model S* with a *CPS problem*: (a) *Deployment View*, there are two processing nodes, e.g., PN_1 and PN_2, with un unbalanced processing, i.e., many tasks (e.g. computation from the software entity instances S_a, S_b, S_c) are assigned to PN_1 whereas PN_2 is not so heavily used (e.g. computation of the software entity instance S_d). The over-utilized processing node will show *high* queue length value, i.e., $\$ql\_1$ (estimated as the maximum value overall its hardware devices, i.e., $\$cpu(i)\_ql$ and $\$disk(j)\_ql$), and a *high* utilization value among its hardware entities either for CPUs, i.e., $\$cpu\_1$ (estimated as the maximum value overall its cpu devices, i.e., $\$cpu(i)\_util$), and disks, i.e., $\$disk\_1$, devices (estimated as the maximum value overall its disk devices, i.e., $\$disk(j)\_util$). The under-utilized processing node will show *low* utilization value among its hardware entities either for cpus, i.e. $\$cpu\_2$, and disks, i.e. $\$disk\_2$, devices. The occurrence of such properties leads to assess that the processing nodes PN_1 and PN_2 originate an instance of the Concurrent Processing Systems antipattern (PN_1, PN_2).

The lower side of Figure 2.3 contains the design changes that can be applied according to the *CPS solution*. The following refactoring actions are represented: (a) the software entity instances must be deployed in a better way, according to the available processing nodes. As consequences of the previous action, if the software instances are deployed in a balanced way then the processing node PN_1 will not be a critical resource anymore, hence $\$ql\_1'$, $\$cpu\_1'$, $\$disk\_1'$ values improves despite the $\$cpu\_2'$, $\$disk\_2'$ values.

The logic formula of the CPS antipattern has been defined in [51] and reported in Equation (2.2), where $\mathbb{P}$ represents the set of all the processing nodes.

$$\begin{aligned} & \exists P_x, P_y \in \mathbb{P} \mid \\ & F_{maxQL}(P_x) \geq Th_{maxQL} \\ \wedge\ & [(F_{maxHwUtil}(P_x, cpu) \geq Th_{maxCpuUtil} \\ \wedge\ & F_{maxHwUtil}(P_y, cpu) < Th_{minCpuUtil}) \\ \vee\ & (F_{maxHwUtil}(P_x, disk) \geq Th_{maxDiskUtil} \\ \wedge\ & (F_{maxHwUtil}(P_y, disk) < Th_{minDiskUtil}))] \end{aligned} \tag{2.2}$$

Table 2.4 reports the functions involved in the CPS specification. The first column of the Table shows the function signatures and the second column provides their descriptions.

Table 2.5 reports the thresholds involved in the CPS specification [51]: all the five thresholds (Th_{maxQL}, $Th_{maxCpuUtil}$, $Th_{minCpuUtil}$, $Th_{maxDiskUtil}$, and $Th_{minDiskUtil}$) are related to performance indices.

Table 2.4: Functions specification for the Concurrent Processing Systems performance antipattern.

Function	Description
float F_{maxQL} (ProcessNode pn_x)	It provides the maximum queue length among the hardware devices hosted by the processing node pn_x.
int $F_{maxHwUtil}$ (ProcessNode pn_x, type T)	It provides the maximum utilization among the hardware devices of a certain type T = {cpu, disk, all} hosted by the processing node pn_x.

Table 2.5: Thresholds specification for the Concurrent Processing Systems performance antipattern.

	Threshold	Description
Performance	Th_{maxQL}	Maximum bound for the hardware device queue length.
	$Th_{maxCpuUtil}$	Maximum bound for the processing device utilization.
	$Th_{minCpuUtil}$	Minimum bound for the processing device utilization.
	$Th_{maxDiskUtil}$	Maximum bound for the disk device utilization.
	$Th_{maxDiskUtil}$	Maximum bound for the disk device utilization.

2.3.3 Pipe & Filter Architectures

Problem - *Occurs when the slowest filter in a "pipe and filter" architecture causes the system to have unacceptable throughput.*

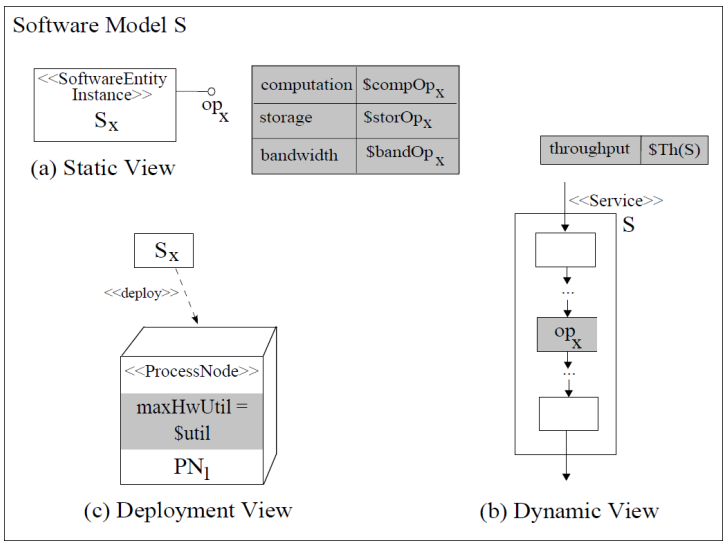
The Pipe and Filter Architectures antipattern (PaF) represents a manifestation of the Unbalanced Processing antipattern [153]. It occurs when the throughput of the overall system is determined by the slowest filter. It means that there is a stage in a pipeline which is significantly slower than all the others, therefore constituting a bottleneck in the whole process in which most stages have to wait the slowest one to terminate.

Solution - Break large filters into more stages and combine very small ones to reduce overhead.

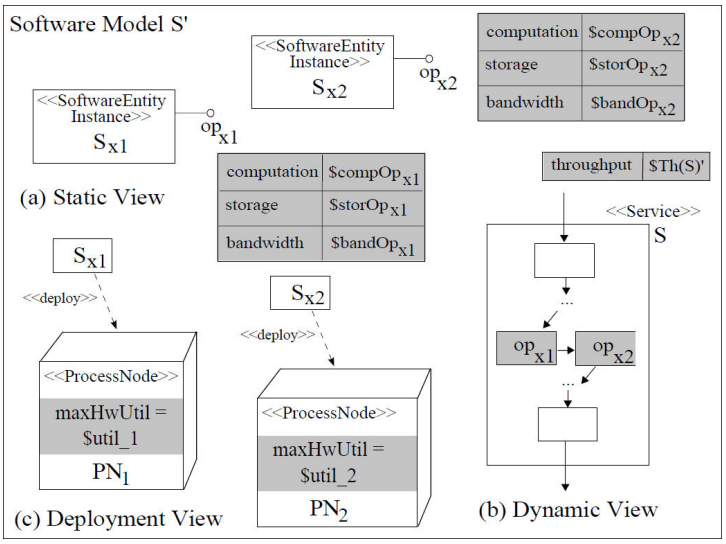
The solution to the Pipe and Filter Architectures antipattern is to 1) divide long processing steps into multiple, smaller stages that can execute in parallel; 2) combine short processing steps to minimize context switching overhead and other delays for shared resources.

Figure 2.4 provides a graphical representation of the Pipe and Filter Architectures antipattern.

The upper side of Figure 2.4 describes the system properties of a *Software Model S* with a *Pipe and Filter Architectures problem*: (a) *Static View*, there is a software entity instance, e.g. S_x, offering an operation (op_x) whose resource demand (computation = $\$compOp_x$, storage = $\$storOp_x$, bandwidth = $\$bandOp_x$) is quite *high*; (b) *Dynamic View*, the operation op_x is invoked in a service S and the throughput



(a) Pipe and Filter Architectures problem.



(b) Pipe and Filter Architectures solution.

Figure 2.4: A graphical representation of the Extensive Processing performance antipattern.

of the service ($\$Th(S)$) is lower than the required one; (c) *Deployment View*, the processing node on which S_x is deployed, i.e. PN_1, might have a *high* utilization value ($\$util$). The occurrence of such properties leads to assess that the operation op_x originates an instance of the Pipe and Filter Architectures antipattern.

The lower side of Figure 2.4 contains the design changes that can be applied according to the *Pipe and Filter Architectures solution*. The following refactoring actions are represented: (a) the operation op_x must be divided in at least two operations, i.e. op_{x1} and op_{x2}, offered by two different software instances; (b) software instances deployed on different processing nodes enable the parallel execution of requests. As consequences of the previous actions, if the operations are executed in parallel then

the operation op_x will not be the slowest filter anymore. The throughput of the service S is expected to improve, i.e. $\$Th(S)' > \$Th(S)$.

The logic formula of the *PaF* performance antipattern has been defined in [51] and reported in Equation (2.3), where $\mathbb{O}$, $\mathbb{S}$, and $\mathrm{sw}\mathbb{E}$, represent the set of all the operations, services, and software entities, respectively.

$$\begin{aligned} &\exists Op \in \mathbb{O}, S \in \mathbb{S} \mid \\ \forall i : F_{resDemand}(Op)[i] \geq Th_{maxOpResDemand}[i] \wedge \\ \wedge \; F_{probExec}(S, Op) = 1 \wedge \\ \wedge \; (F_{maxHwUtil}(P_{swE_x}, all) \geq Th_{maxHwUtil} \vee \\ \vee \; F_T(S) < Th_{SthReq}) \end{aligned} \tag{2.3}$$

Table 2.6 reports the functions involved in the PaF specification. The first column of the Table shows the function signatures and the second column provides their descriptions.

Table 2.6: Functions specification for the Pipe and Filter Architectures performance antipattern.

Function	Description
float[] $F_{resDemand}$ (Operation Op)	It provides the resource demand of the operation Op.
int $F_{probExec}$ (Service S, Operation Op_i)	It provides the probability the operation Op is executed in the service S.
float $F_{maxHwUtil}$ (ProcessNode pn_x, type T)	It provides the maximum utilization among the hardware devices of a certain type T = {cpu, disk, all} hosted by the processing node pn_x.
float F_T (Service S)	It provides the estimated throughput of the service S at the steady-state.

Table 2.7 reports the thresholds involved in the PaF specification [51]: all the three thresholds ($Th_{maxOpResDemand}[j]$, $Th_{maxHwUtil}$, and Th_{SthReq}) are related to performance indices.

Table 2.7: Thresholds specification for the Pipe and Filter Architectures performance antipattern.

	Threshold	Description
Performance	$Th_{maxOpResDemand}[j]$	Maximum bound for the resource demand of operations.
	$Th_{maxHwUtil}$	Maximum bound for the hardware device utilization.
	Th_{SthReq}	Required value for the throughput of the service S.

2.3.4 Extensive Processing

Problem *- Occurs when extensive processing in general impedes overall response time.*

The Extensive Processing antipattern represents a manifestation of the Unbalanced Processing antipattern [153]. It occurs when a long running process monopolizes a processor and prevents a set of other jobs to be executed until it finishes its computation. The processor is removed from the pool, but other work does not have to pass through this stage before proceeding. This is particularly problematic if the extensive processing is on the processing path that is executed for the most frequent workload.

Solution - Move extensive processing so that it does not impede high traffic or more important work.

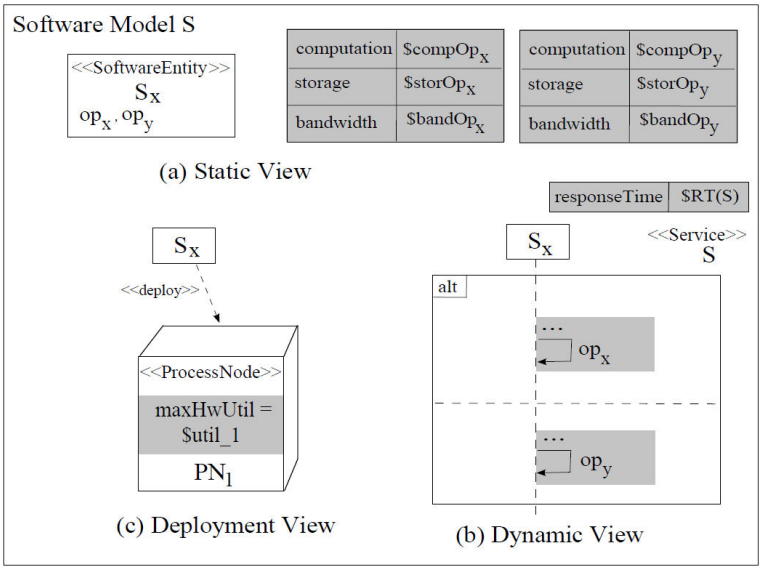
The solution to the Extensive Processing antipattern is to identify processing steps that may cause slow downs and delegate those steps to processes that will not impede the fast path. A performance improvement could be achieved by delegating processing steps which do not need a synchronous execution to other processes.

Figure 2.5 graphically shows the Extensive Processing antipattern.

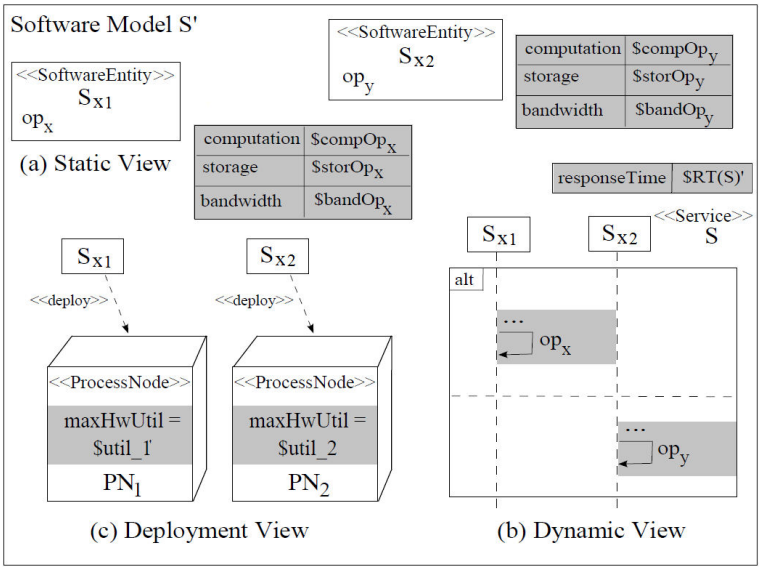
The upper side of Figure 2.5 describes the system properties of a *Software Model S* with an *EP problem*: (a) *Static View*, there is a software entity instance, e.g. S_x, having two operations (op_x, op_y) whose resource demand is quite unbalanced, since op_x has a high demand (see the MARTE annotation *computation* = $\$compOp_x$, *storage* = $\$storOp_x$, *bandwidth* = $\$bandOp_x$), whereas op_y has a low demand (see *computation* = $\$compOp_y$, *storage* = $\$storOp_y$, *bandwidth* = $\$bandOp_y$); (b) *Dynamic View*, the operations op_x and op_y are alternatively invoked in a service S, and the response time of the service ($\$RT(S)$) is larger than the required one; (c) *Deployment View*, the processing node on which S_x is deployed, i.e. PN_1, might reveal a high utilization value ($\$util$). The occurrence of such properties leads to assess that the operations op_x and op_y originate an instance of the EP antipattern.

The lower side of Figure 2.5 contains the design changes that can be applied according to the *EP solution*. The following refactoring actions are represented: (a) the operations op_x and op_y must be owned by two different software instances; (b) software instances deployed on different processing nodes provide a fast path for requests. As consequences of the previous actions, the response time of the service S is expected to improve, i.e. $\$RT(S)' < \$RT(S)$ [110].

The logic formula of the EP has been defined in [51] and reported in Equation (2.4), where $\mathbb{O}$, $\mathbb{S}$, and $\mathrm{sw}\mathbb{E}$, represent the set of all the operations, services, and software entities, respectively.



(a) Extensive Processing problem.



(b) Extensive Processing solution.

Figure 2.5: A graphical representation of the Extensive Processing performance antipattern.

$$
\begin{aligned}
&\exists Op_1, Op_2 \in \mathbb{O}, S \in \mathbb{S}, swE_x \in \mathrm{sw}\mathbb{E} \mid \\
&\forall i : F_{resDemand}(Op_1)[i] \geq Th_{maxOpResDemand}[i] \wedge \\
\wedge\ &\forall i : F_{resDemand}(Op_2)[i] < Th_{minOpResDemand}[i] \wedge \\
\wedge\ &F_{probExec}(S, Op_1) + F_{probExec}(S, Op_2) = 1 \wedge \\
\wedge\ &(F_{maxHwUtil}(P_{swE_x}, all) \geq Th_{maxHwUtil} \vee \\
\vee\ &F_{RT}(S) > Th_{sRtReq})
\end{aligned}
\tag{2.4}
$$

Table 2.8 reports the functions involved in the EP specification. The first column of the Table shows the function signatures and the second column provides their descriptions.

Table 2.8: Functions specification for the Extensive Processing performance antipattern.

Function	Description
float[] $F_{resDemand}$ (Operation Op)	It provides the resource demand of the operation Op.
int $F_{probExec}$ (Service S, Operation Op_i)	It provides the probability the operation Op is executed in the service S.
float $F_{maxHwUtil}$ (ProcessNode pn_x, type T)	It provides the maximum utilization among the hardware devices of a certain type T = {cpu, disk, all} hosted by the processing node pn_x.
float F_{RT} (Service S)	It provides the estimated response time of the service S at the steady-state.

Table 2.9 reports the thresholds involved in the EP specification [51]: all the four thresholds ($Th_{maxOpResDemand}[j]$, $Th_{minOpResDemand}[j]$, $Th_{maxHwUtil}$, and Th_{sRtReq}) are related to performance indices.

Table 2.9: Thresholds specification for the Extensive Processing performance antipattern.

	Threshold	Description
Performance	$Th_{maxOpResDemand}[j]$	Maximum bound for the resource demand of operations.
	$Th_{minOpResDemand}[j]$	Minimum bound for the resource demand of operations.
	$Th_{maxHwUtil}$	Maximum bound for the hardware device utilization.
	Th_{sRtReq}	Required value for the response time of the service S.

2.3.5 Empty Semi-Trucks

Problem *- Occurs when an excessive number of requests is required to perform a task. It may be due to inefficient use of available bandwidth, an inefficient interface, or both.*

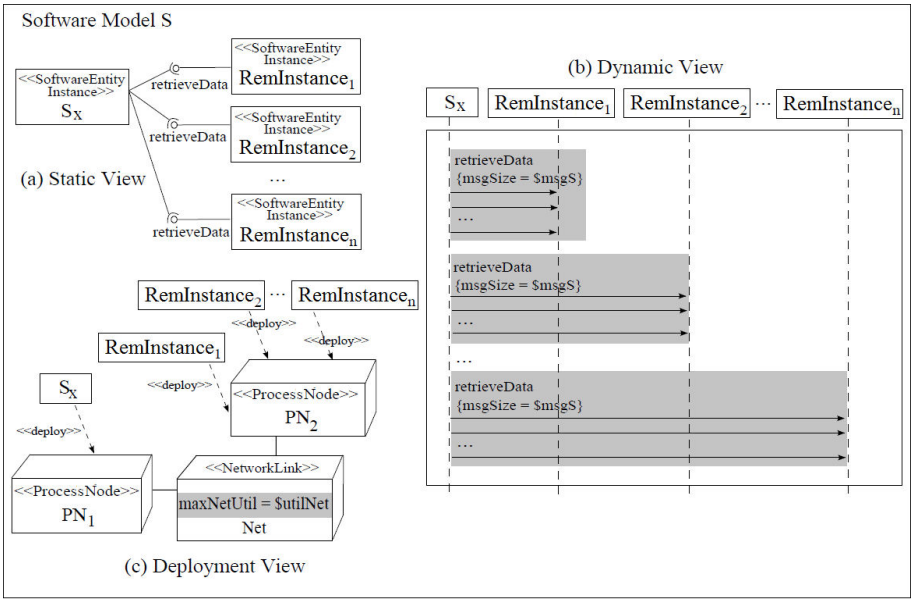
The problem of inefficient use of available bandwidth typically affects message-based systems when a huge load of messages, each containing a small amount of information, is exchanged over the network. The amount of processing overhead is the same regardless of the size of the message. With smaller messages, this processing is required many more times than necessary hence it significantly implies a performance loss. The problem of an inefficient interface (i.e. it provides a too fragmented access to data) generates an excessive overhead caused by the computation needed to handle each call request.

Solution *- The Session Facade design pattern to provide more efficient interfaces and a better use of available bandwidth [159].*

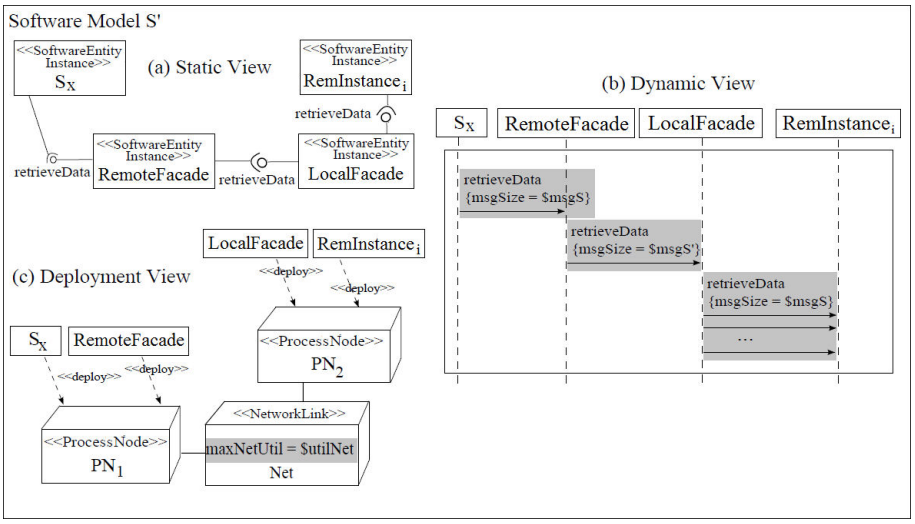
A possible solution is given by the adoption of the Session Facade design pattern [159], which introduces two new software entities for mediating the communication

between the ones involved in the EST problem: requests are grouped in larger chunks to minimize the overhead due to information spread and processing. In this way, several messages are merged into a single bigger message, resulting in time savings.

Figure 2.6 provides a graphical representation of the *Empty Semi Trucks* antipattern.



(a) Empty Semi-Trucks problem.



(b) Empty Semi-Trucks solution.

Figure 2.6: A graphical representation of the Empty Semi-Trucks performance antipattern.

The upper side of Figure 2.6 describes the system properties of a *Software Model S* with a *Empty Semi Trucks problem*: (a) *Static View*, there is a software entity instance, e.g. S_x, retrieving some information from several instances ($RemInstance_1$, ..., $RemInstance_n$); (b) *Dynamic View*, the software instance S_x generates an *excessive* message traffic by sending a big amount of messages by *low* sizes ($\$msgS$),

much lower than the network bandwidth, hence the network link might have a *low* utilization value ($\$utilNet$); (c) *Deployment View*, the processing node on which S_x is deployed, i.e. PN_1, might reveal a *high* utilization value ($\$util$). The occurrence of such properties leads to assess that the software instance S_x originates an instance of the EST antipattern.

The lower side of Figure 2.6 contains the design changes that can be applied according to the *Empty Semi Trucks solution*. The following refactoring action is represented: (a) *Static View*, the introduction of two new software entity instances, i.e. $RemoteFacade$ and $LocalFacade$, between S_x and the remote instances $RemInstance_i$; (b) *Dynamic View*, the communication between S_x and the remote instances $RemInstance_i$ must be restructured in such a way that multiple requests to the same $RemInstance_i$ are grouped in a single request by S_x that is then brought to $RemInstance_i$ through $RemoteFacade$ and $LocalFacade$, in order to reduce the number of messages through Net; (c) *Deployment View*, $RemoteFacade$ and $LocalFacade$ are deployed on PN_1 and PN_2 respectively, to make the former local to S_x and the latter local to $RemInstance_i$.

The logic formula of the *EST* has been defined in [51] and reported in Equation (2.5), where $\mathbb{S}$ and $\mathrm{sw}\mathbb{E}$ represent the set of all the services and software entities, respectively. Each (swE_x, S) instance satisfying the predicate must be pointed out to the designer for a deeper analysis, because it represents an EST occurrence.

$$\begin{aligned} &\exists swE_x \in \mathrm{sw}\mathbb{E}, S \in \mathbb{S} \mid \\ &F_{numRemMsgs}(swE_x, S) \geq Th_{maxRemMsgs} \wedge \\ \wedge\ &(F_{maxNetUtil}(P_{swE_x}, swE_x) < Th_{minNetUtil} \vee \\ &F_{numRemInst}(swE_x, S) \geq Th_{maxRemInst}) \end{aligned} \tag{2.5}$$

Table 2.10 reports the functions involved in the EST specification. The first column of the Table shows the function signatures and the second column provides their descriptions.

Table 2.10: Functions specification for the Empty Semi-Trucks performance antipattern.

Function	Description
int $F_{numRemMsgs}$ (SoftwareEntityInstance swE_x, Service S)	It counts the number of remote messages sent by swE_x in a service S.
float $F_{maxNetUtil}$ (ProcessNode pn_x, SoftwareEntityInstance swE_x)	It provides the maximum utilization among the network links connecting pn_x overall the processing nodes with which swE_x generates traffic.
int $F_{numRemInst}$ (SoftwareEntityInstance swE_x, Service S)	It provides the number of remote instances with which swE_x communicates in a service S.

Table 2.11 reports the thresholds involved in the EST specification [51]: two thresholds ($Th_{maxRemMsgs}$, $Th_{maxRemInst}$) refer to design features, whereas the third one ($Th_{minNetUtil}$) is related to a performance index.

Table 2.11: Thresholds specification for the Empty Semi-Trucks performance antipattern.

	Threshold	Description
Design	$Th_{maxRemMsgs}$	Maximum bound for the number of remote messages in a service.
	$Th_{maxRemInst}$	It represents the maximum bound for the number of remote communicating instances in a service.
Performance	$Th_{minNetUtil}$	Minimum bound for the network link utilization.

2.3.6 Traffic Jam

Problem *- Occurs when one problem causes a backlog of jobs that produces wide variability in response time which persists long after the problem has disappeared.*

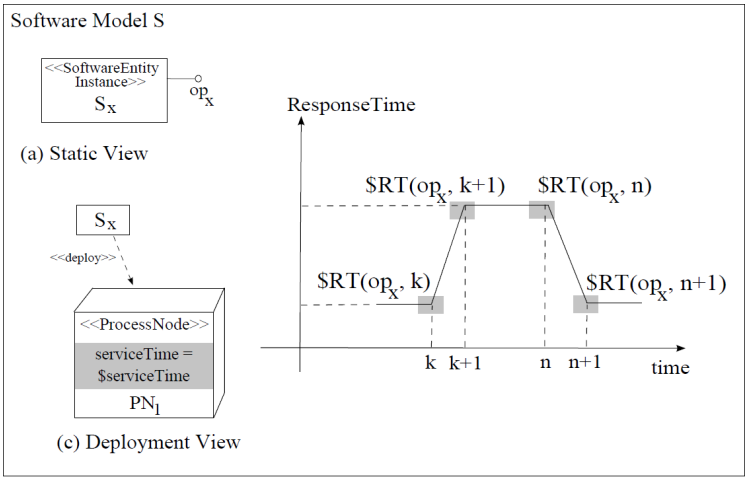
The Traffic Jam antipattern occurs if a significative variability in response time is observed, and it is due to different causes, usually difficult to identify. The problem also occurs when a large amount of work is scheduled within a relatively small interval. It occurs, for example, when a huge number of processes are originated at approximately the same time. The challenge is that often the performance loss and the slowing down of the computation arises and persists for a long time after the originating cause has disappeared. A failure or any other cause of performance bottleneck generate a backlog of jobs waiting for computation, which take a long time to be executed and to return in a normal operating condition.

Solution *- Begin by eliminating the original cause of the backlog. If this is not possible, provide sufficient processing power to handle the worst-case load.*

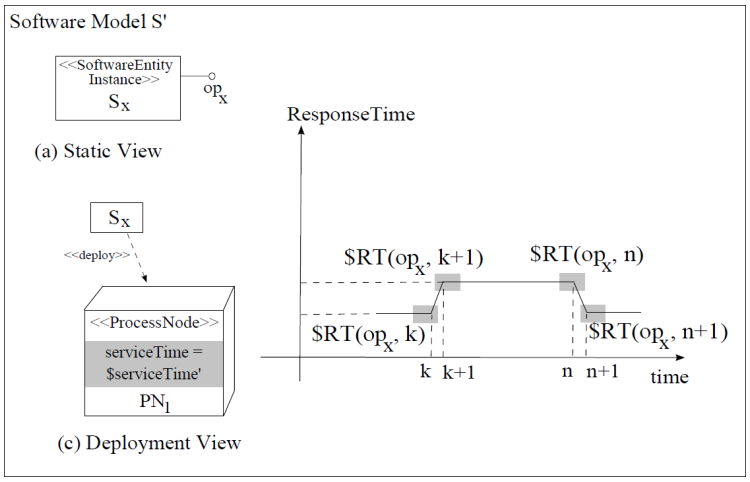
If the problem is caused by periodic high demand, it might be beneficial to seek alternatives that spread the load, or handle the demand in a different manner. For example, if users select the time for performing a request to the system, it may be helpful to change the selection options so that they select a time interval rather than selecting a specific time. It gives the software more flexibility in scheduling the requests in order to reduce contention. If the problem is caused by external factors, such as domain specific behaviors, than it is important to determine the size of the platforms and networks that will be required to support the worst-case workload intensity. The ideal solution is clearly given by the elimination of the originating bottleneck. However, localizing the source of the backlog is not simple and the increase of the computing power, even costly, could benefit.

Figure 2.7 and provides a graphical representation of the Traffic Jam antipattern.

The upper side of Figure 2.7 describes the system properties of a *Software Model S* with a *Traffic Jam problem*: (a) *Static View*, there is a software entity instance S_x offering an operation op_x; the monitored response time of the operation op_x shows "*a wide variability in response time which persists long* " [152]. The occurrence of such properties leads to assess that the software instance S_x originates an instance of the Traffic Jam antipattern.



(a) Traffic Jam problem.



(b) Traffic Jam solution.

Figure 2.7: A graphical representation of the Traffic Jam performance antipattern.

The lower side of Figure 2.7 contains the design changes that can be applied according to the *Traffic Jam solution*. The following refactoring action is represented: (a) the service time of the processing node on which S_x is deployed must be decreased, i.e. $\$serviceTime'$. As consequences of the previous action, the response time of the operation op_x is expected to increase in a slower way.

The logic formula of the Traffic Jam antipattern has been defined in [51] and reported in Equation (2.6), where $\mathbb{O}$ represents the set of all operation instances.

$$\begin{aligned} & \exists OpI \in \mathbb{O} \mid \\ & \frac{\sum_{1 \leq t \leq k} \mid (F_{RT}(OpI,t) - F_{RT}(OpI,t-1)) \mid}{k-1} < Th_{OpRtVar} \\ \wedge \quad & \mid F_{RT}(OpI,k) - F_{RT}(OpI,k-1) \mid > Th_{OpRtVar} \\ \wedge \quad & \frac{\sum_{k \leq t \leq n} \mid (F_{RT}(OpI,t) - F_{RT}(OpI,t-1)) \mid}{n-k} < Th_{OpRtVar} \end{aligned} \tag{2.6}$$

Table 2.12 reports the functions involved in the Traffic Jam specification. The first column of the Table shows the function signatures and the second column provides their descriptions.

Table 2.12: Functions specification for the Traffic Jam performance antipattern.

Function	Description
float F_{RT} (OperationInstance OpI, timeInterval t)	It provides the estimated response time of the operation instance OpI at the time interval t.

Table 2.13 reports the thresholds involved in the Traffic Jam specification [51]: the unique threshold ($Th_{OpRtVar}$) is related to a performance index.

Table 2.13: Threshold specification for the Traffic Jam performance antipattern.

	Threshold	Description
Performance	$Th_{OpRtVar}$	Maximum bound for the variability in response times of operations across simulation intervals.