Understanding Performance Aspects of Layered Software with Layered Resources

Murray Woodside
Department of Systems and Computer Engineering
Carleton University, Ottawa, Canada
cmw@sce.carleton.ca, www.layered queues.org

Jan.27, 2003

Carleton UNIVERSITY

Carleton UNIVERSITY

The Challenge of Performance in Distributed and Parallel Software

- several programs interact to complete one response
  - clients and servers..... peers.... pipelines
- they execute partly in sequence and partly in parallel
- ***models are required***
  - systems are difficult to measure in the lab (too big)
- performance is governed by many kinds of factors:
  - congestion at different kinds of resources
  - layering of resources
  - layered system overheads
  - unbalanced parallel paths

Carleton UNIVERSITY

Advantages of the layered queueing approach

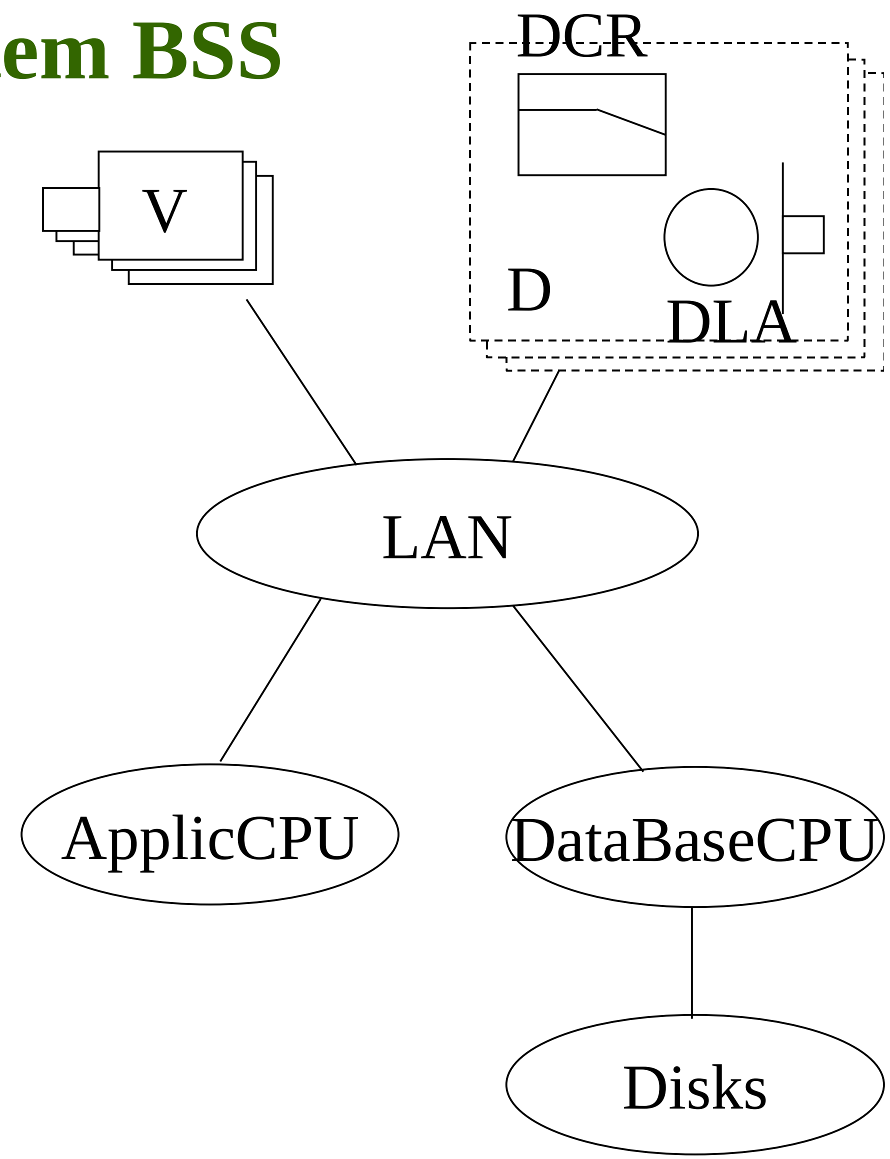
- an elegant formulation of extended queueing for layered resources
  - scales up to many resources and complex holding patterns
- layered resources have understandable patterns
  - layered resources are common...
  - client-server, parallel service, pipeline, and others
  - bottleneck patterns
- the model notation resembles software design notations such as UML
  - UML performance profile can provide parameter annotations for scenarios in UML.

Carleton UNIVERSITY

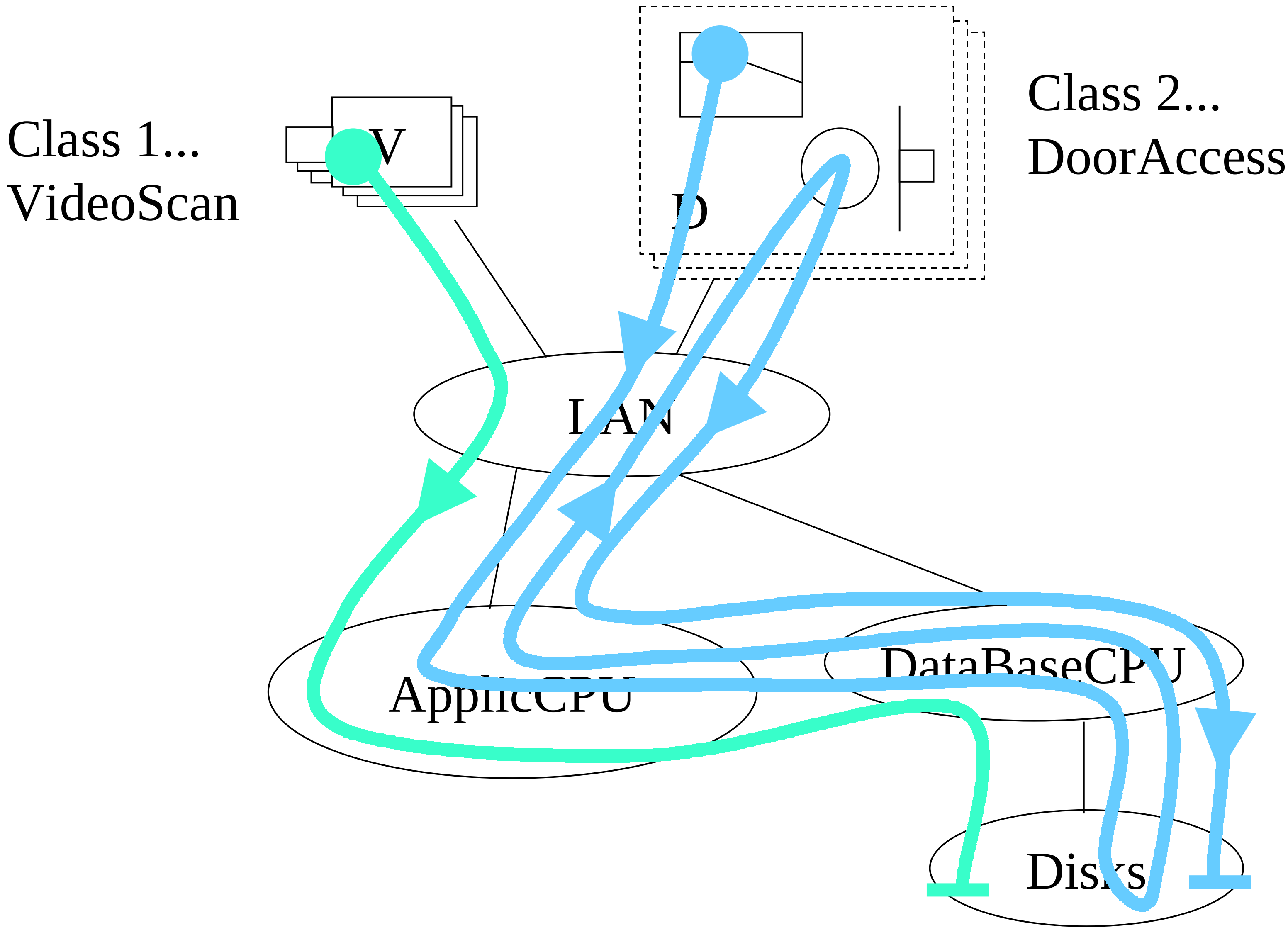
Example: layered modeling of a Building Security System BSS

.....e.g., for a hotel or a university building

- *video surveillance*:
  - poll N web cameras
  - 1 second cycle (on 95% of polling cycles)
- *door access*:
  - respond to an access card within 1 second, 95% of the time
  - card reader DCR, lock actuator DLA

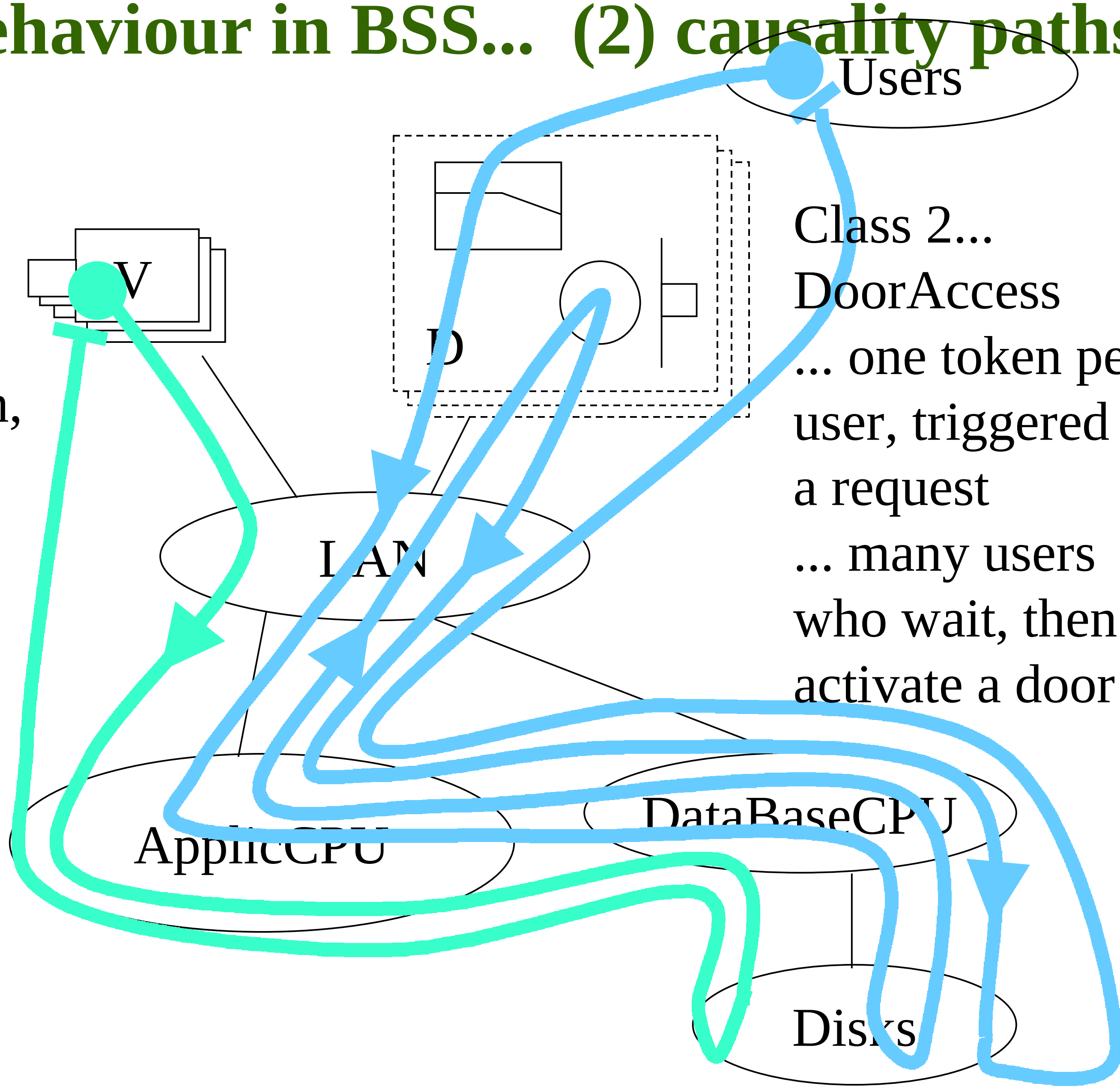


System behaviour in BSS.... (1) data paths



System behaviour in BSS... (2) causality paths

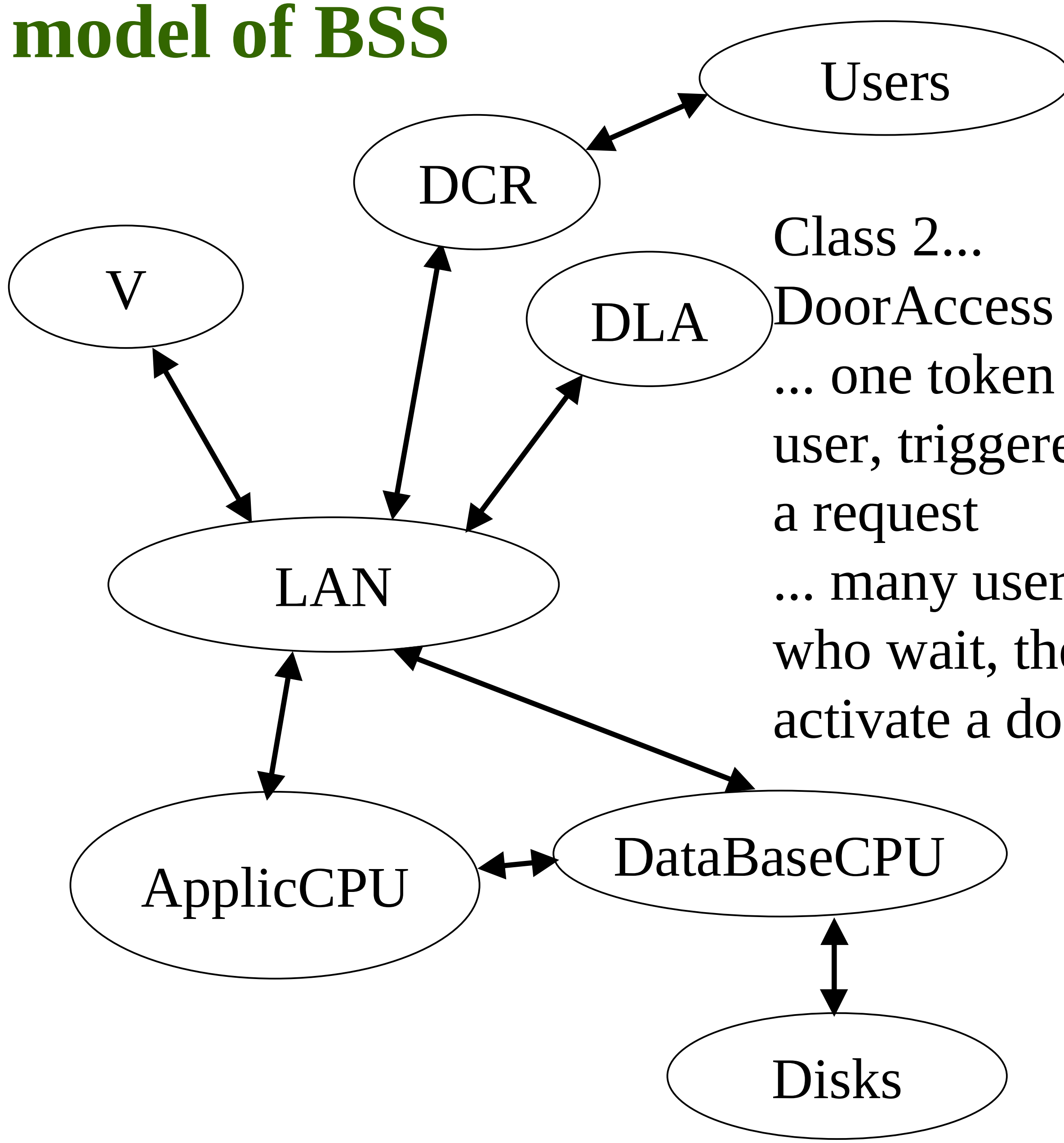
Class 1... VideoScan ... one token, cycling through the cameras



Class 2... DoorAccess ... one token per user, triggered by a request ... many users who wait, then activate a door

A queueing model of BSS

Class 1... VideoScan ... one token, cycling through the cameras ... or more than one, for double buffering



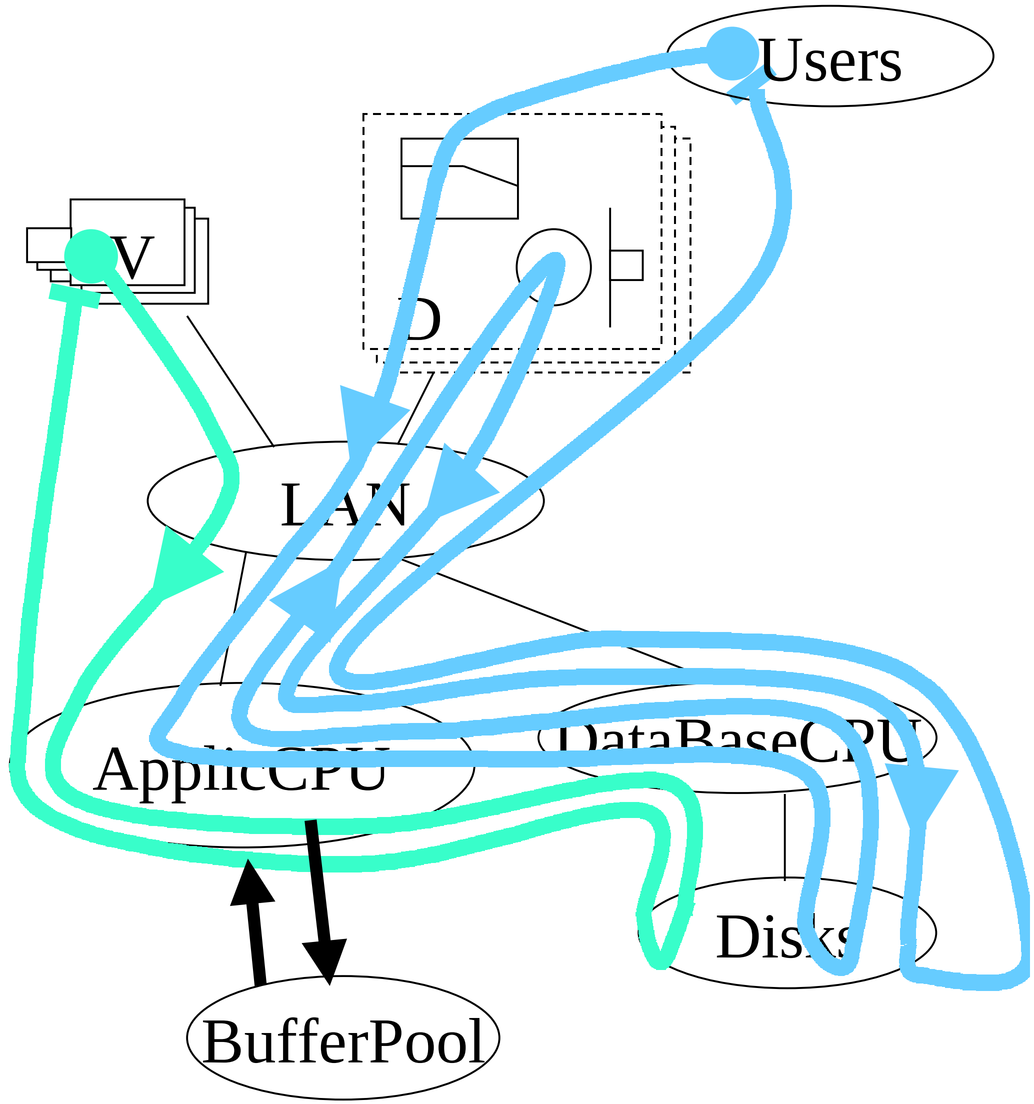
Class 2... DoorAccess ... one token per user, triggered by a request ... many users who wait, then activate a door

- *focuses on hardware servers only*

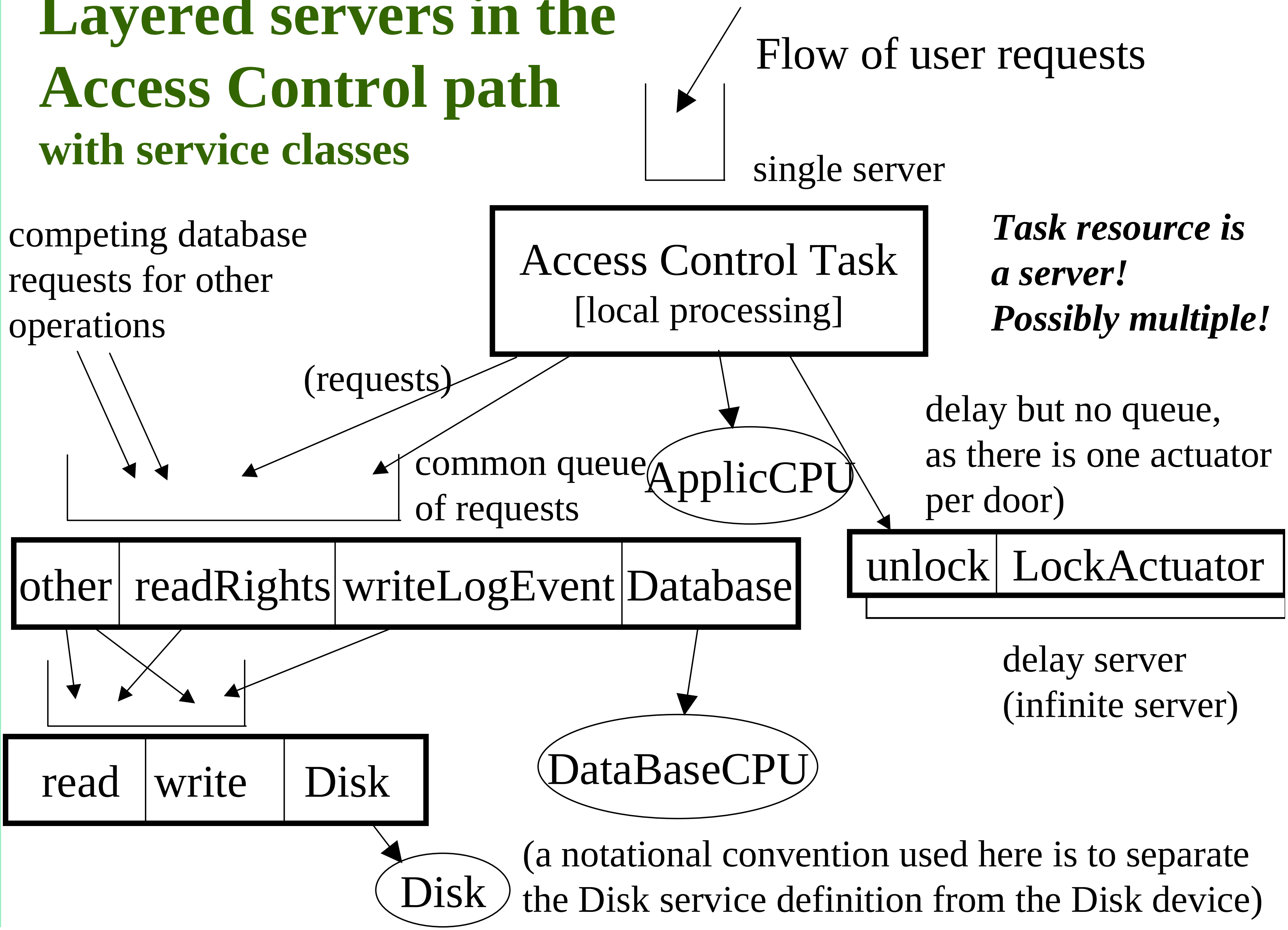
Carleton UNIVERSITY

Additional resources that impact performance

- multiple ***buffers***, size of buffer pool
- ***access control task*** can have a message queue
- ***database task*** may have a queue of requests
- ***software resources***
- ***layered service***



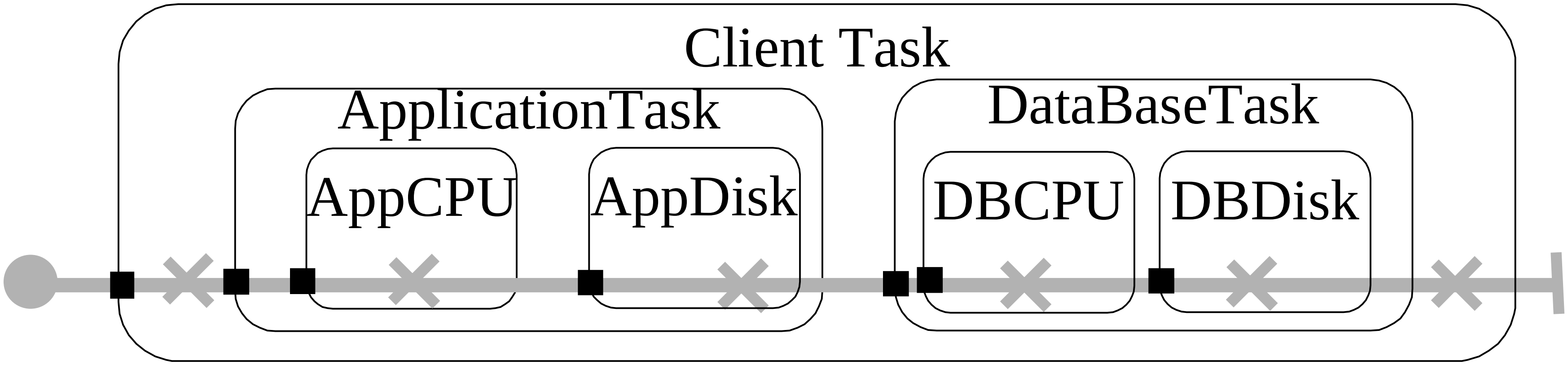
Layered servers in the Access Control path
with service classes



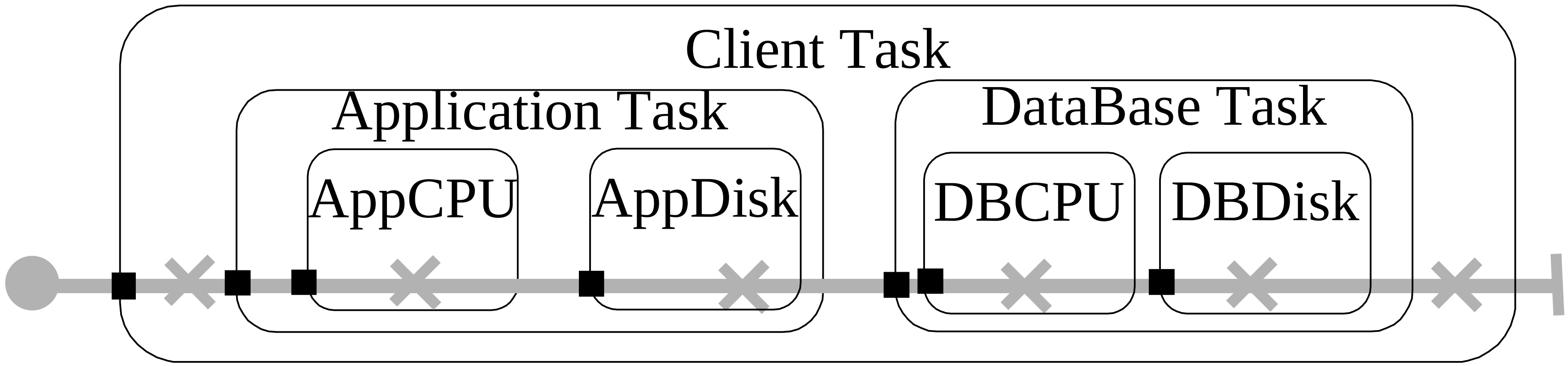
Carleton UNIVERSITY

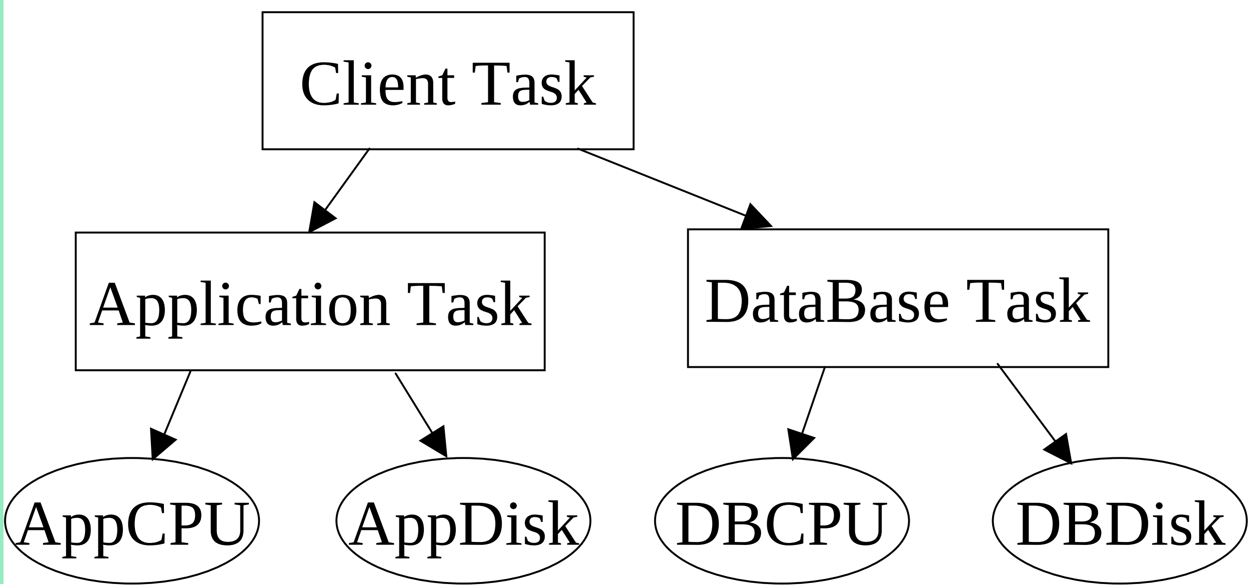
Analysis must include the *software description*

- software operations
- resources that they require
  - task thread resources, buffers, devices
- a useful concept is the *resource context* of an operation:
  - its context is the set of operations during which the resource must be held
  - e.g., outside rectangle represents a client task resource
    - within this context it acquires other resources



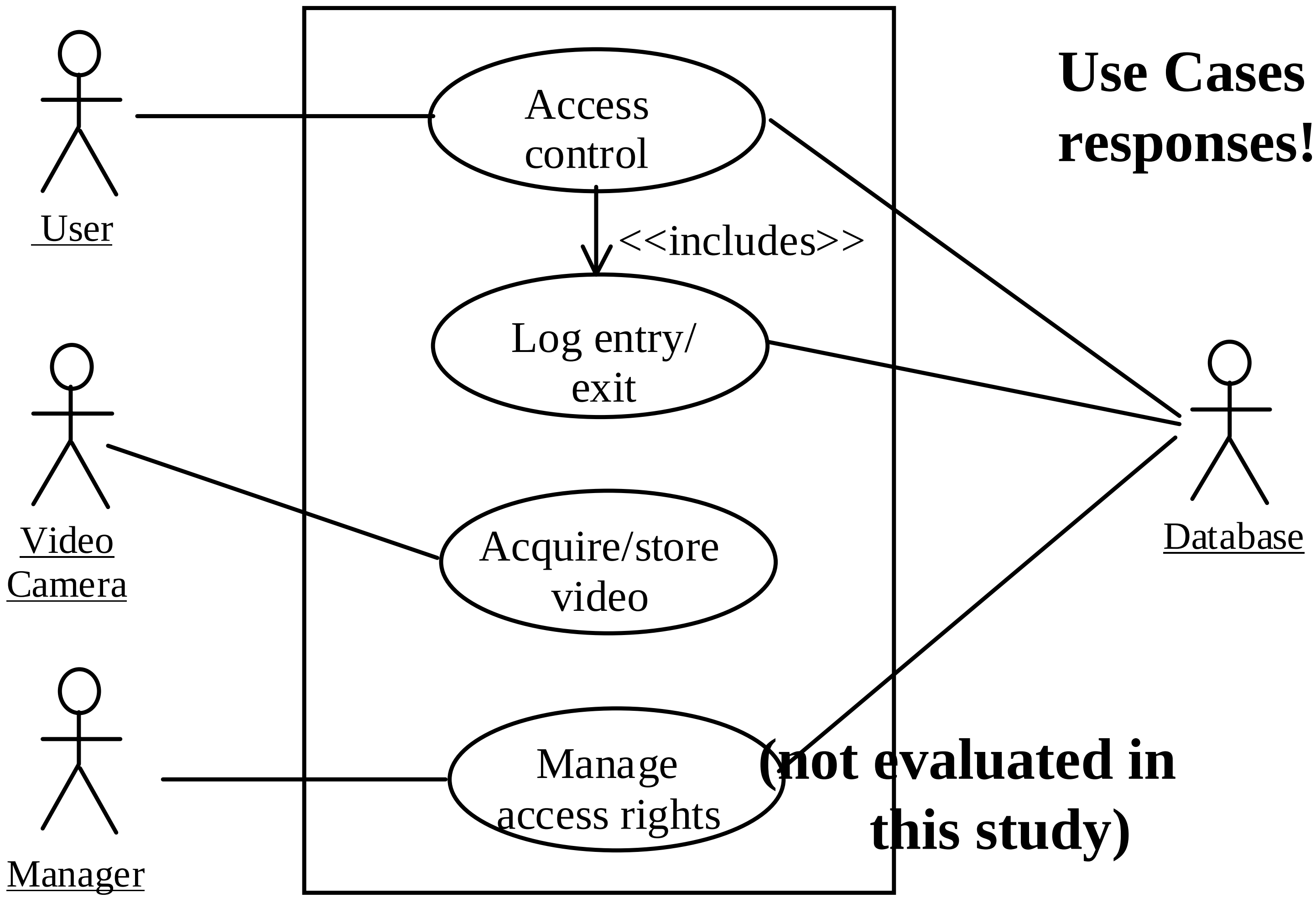
Nested resource contexts gives layered resources





- like a procedure call tree

Software Specification of the Building Security System (BSS): (1) Use Cases

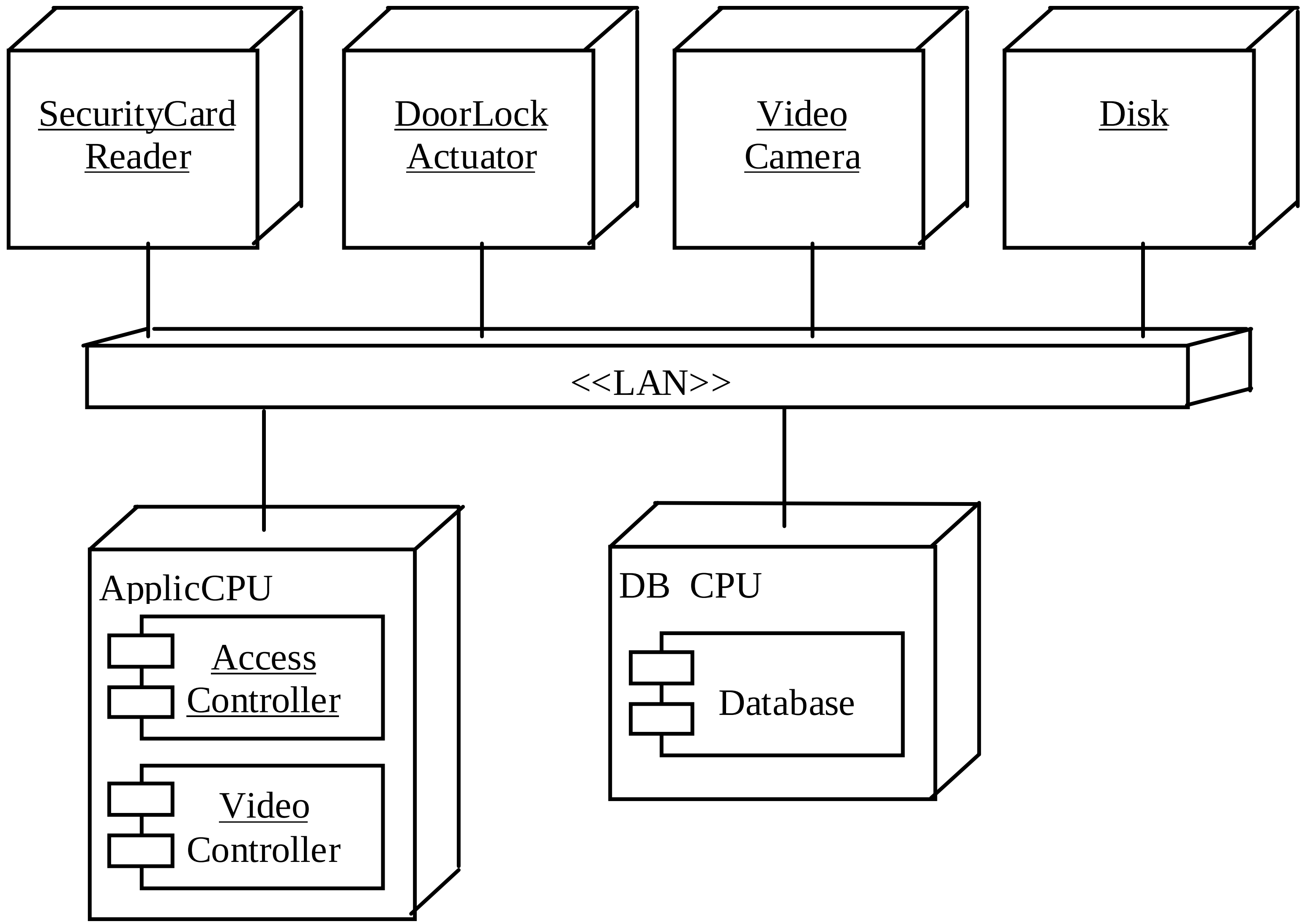


Use Cases define responses!

(not evaluated in this study)

Carleton UNIVERSITY

Software Specification (2) Deployment

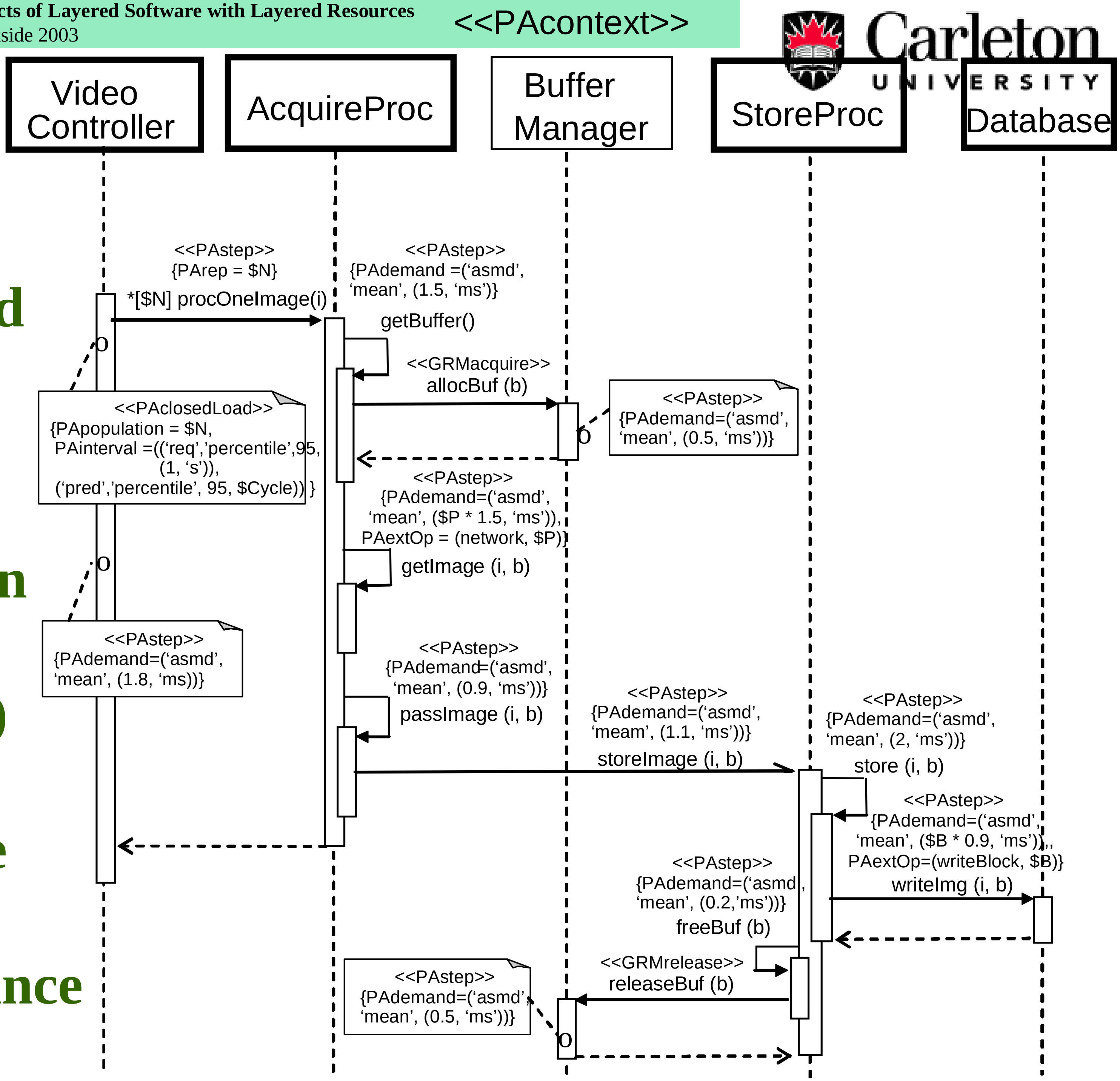


Carleton UNIVERSITY

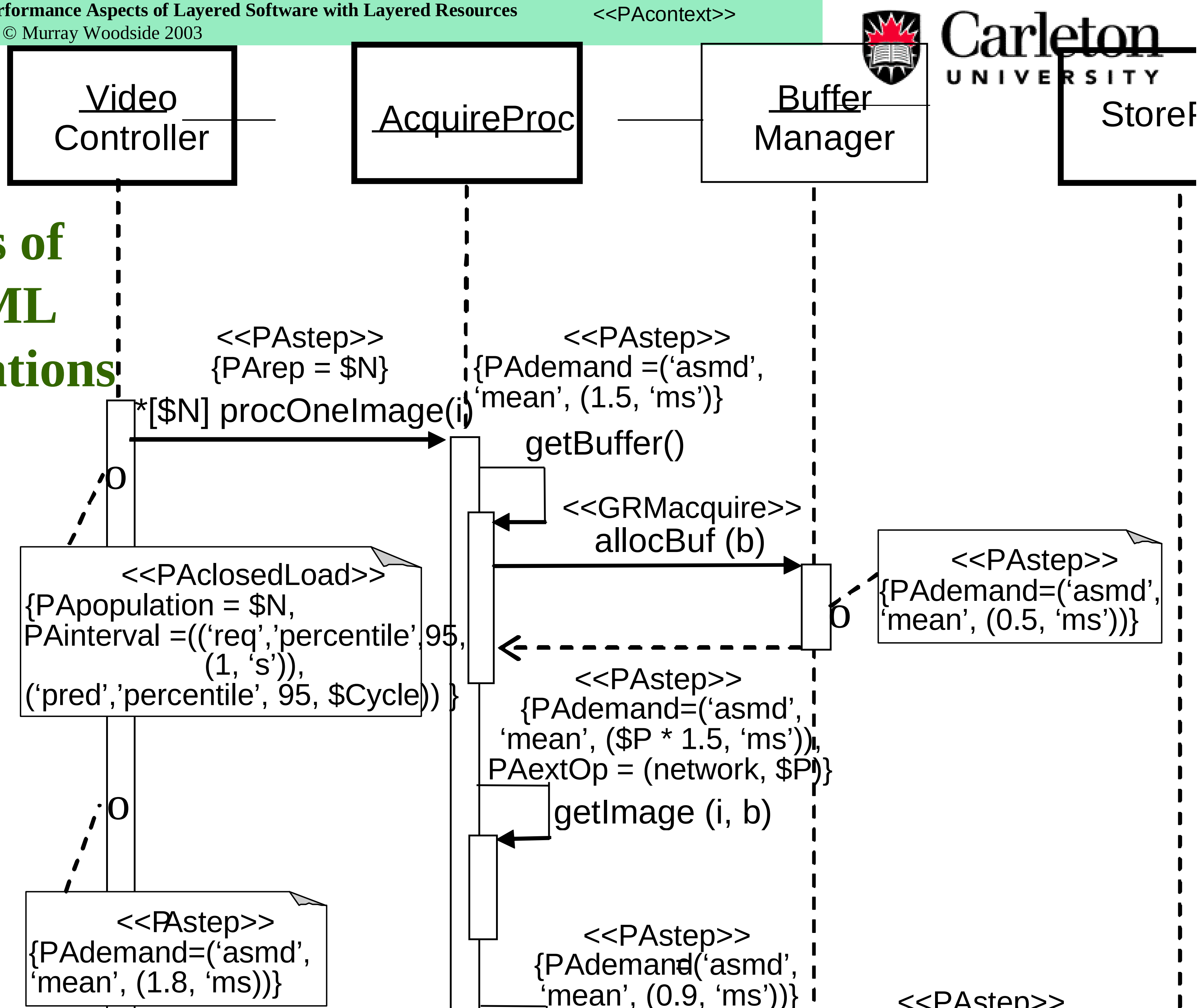
Spec. of BSS (3):

Annotated Sequence Diagram for the VideoScan Use Case (scenario)

(using the UML Performance Profile)

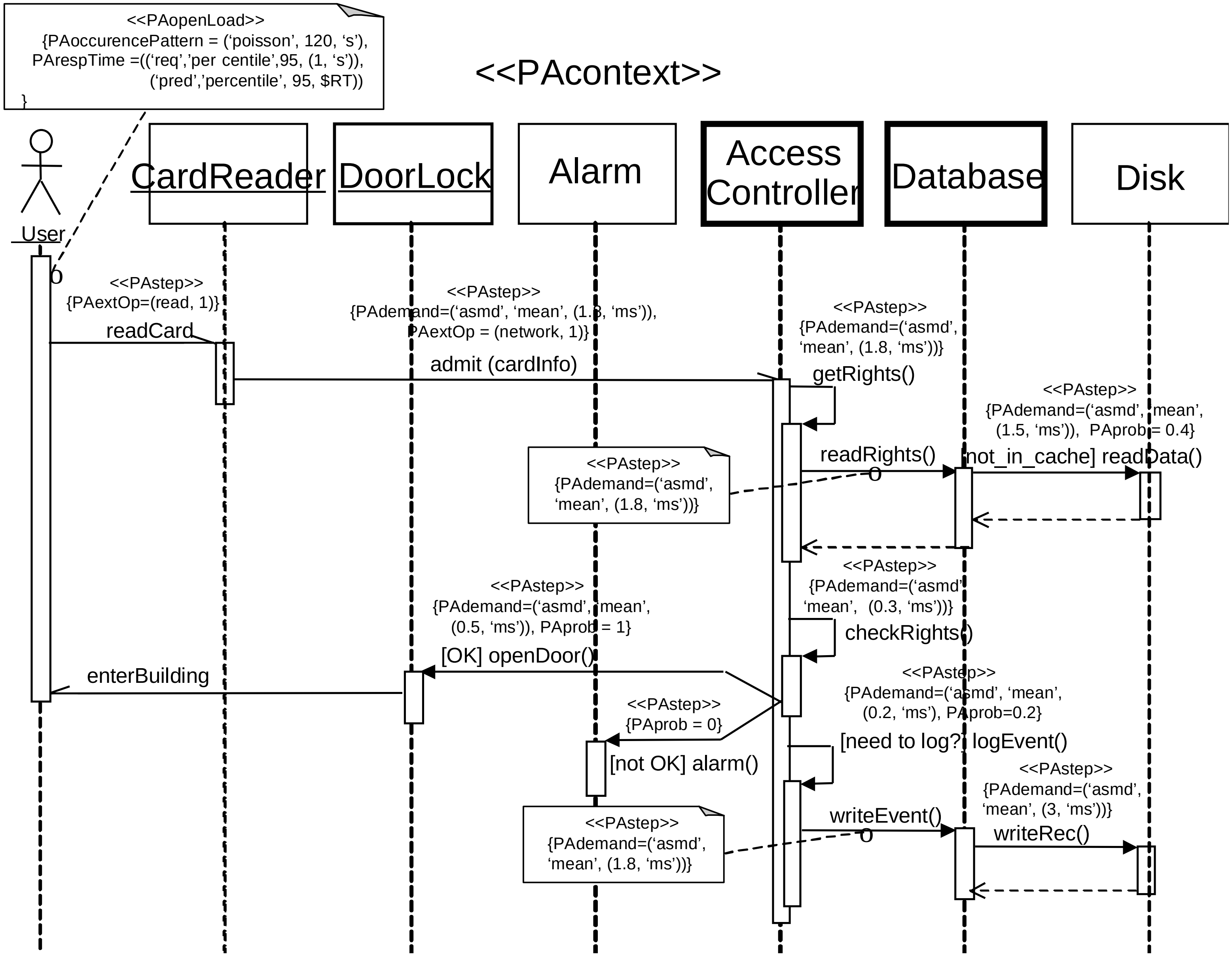


Details of the UML annotations

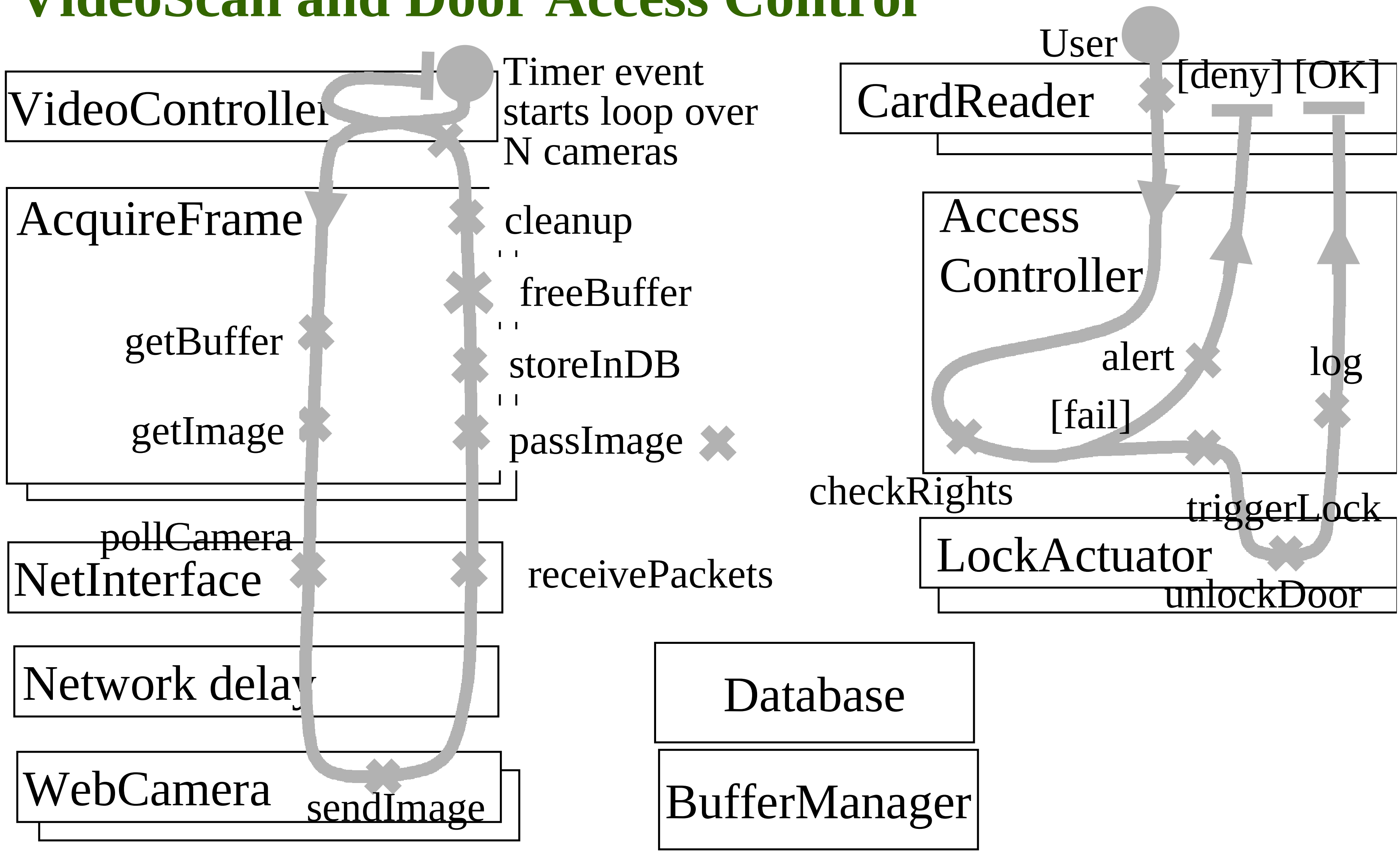


Carleton UNIVERSITY

Spec of BSS (4) Sequence Diagram for the Door Access Control scenario



Simplified view of the main scenarios: VideoScan and Door Access Control



AcquireFrame can use multiple buffers to receive and store images

Resources are identified from annotations

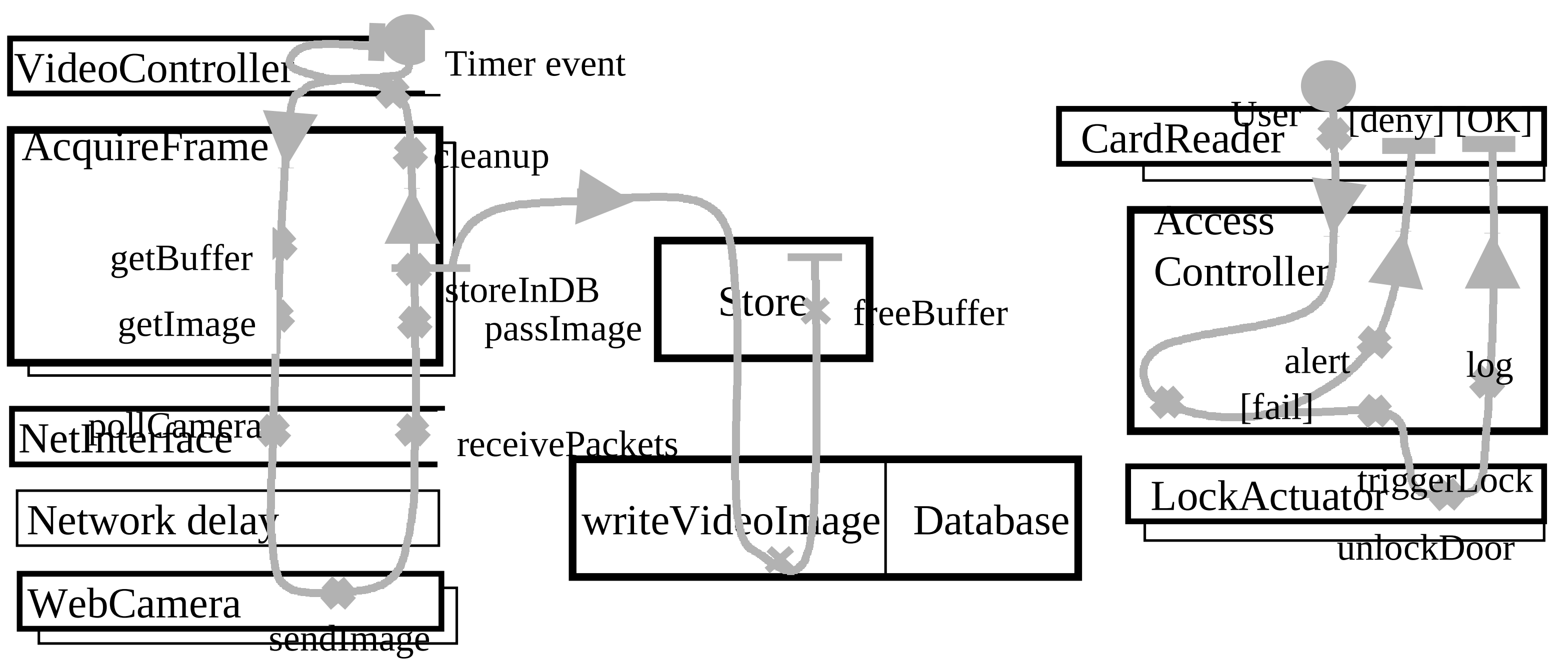
- directly...
  - "active object" or active component == a task
  - deployment of tasks
  - task allocation of component
  - logical resource acquire/release stereotypes

... and as attributes of the activities and components

  - processor for an activity
  - database queried by an activity
  - process or task containing a component
  - network conveying messages between components
  - buffer acquired by an activity getBuffer, released

Task resources: AcquireFrame, and AccessController

- If ***AcquireFrame*** is single-threaded it sequentializes the acquisition
- double buffering requires another thread, or a concurrent task, let us call it Store

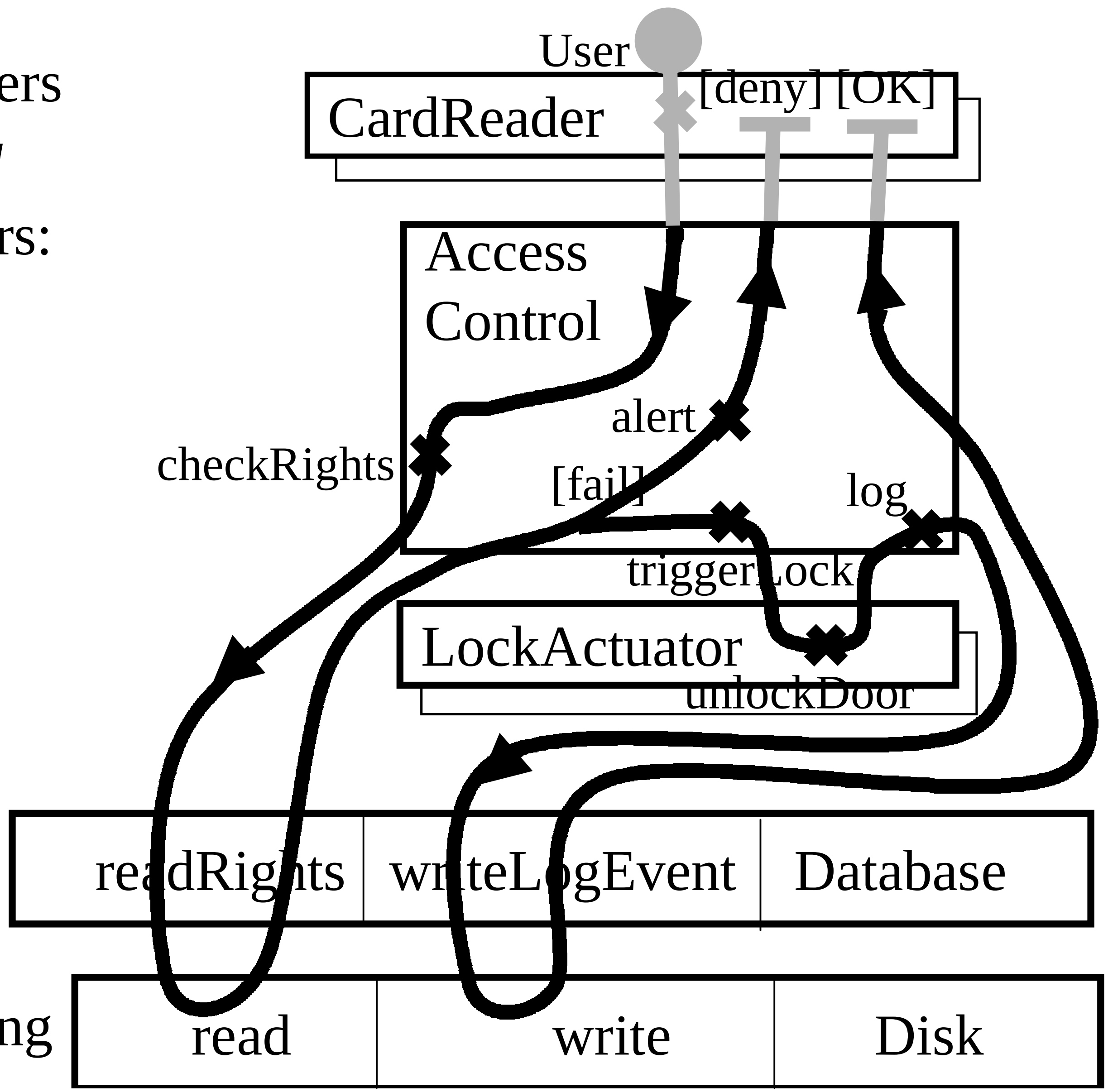


Carleton UNIVERSITY

The service time of a task covers its resource context

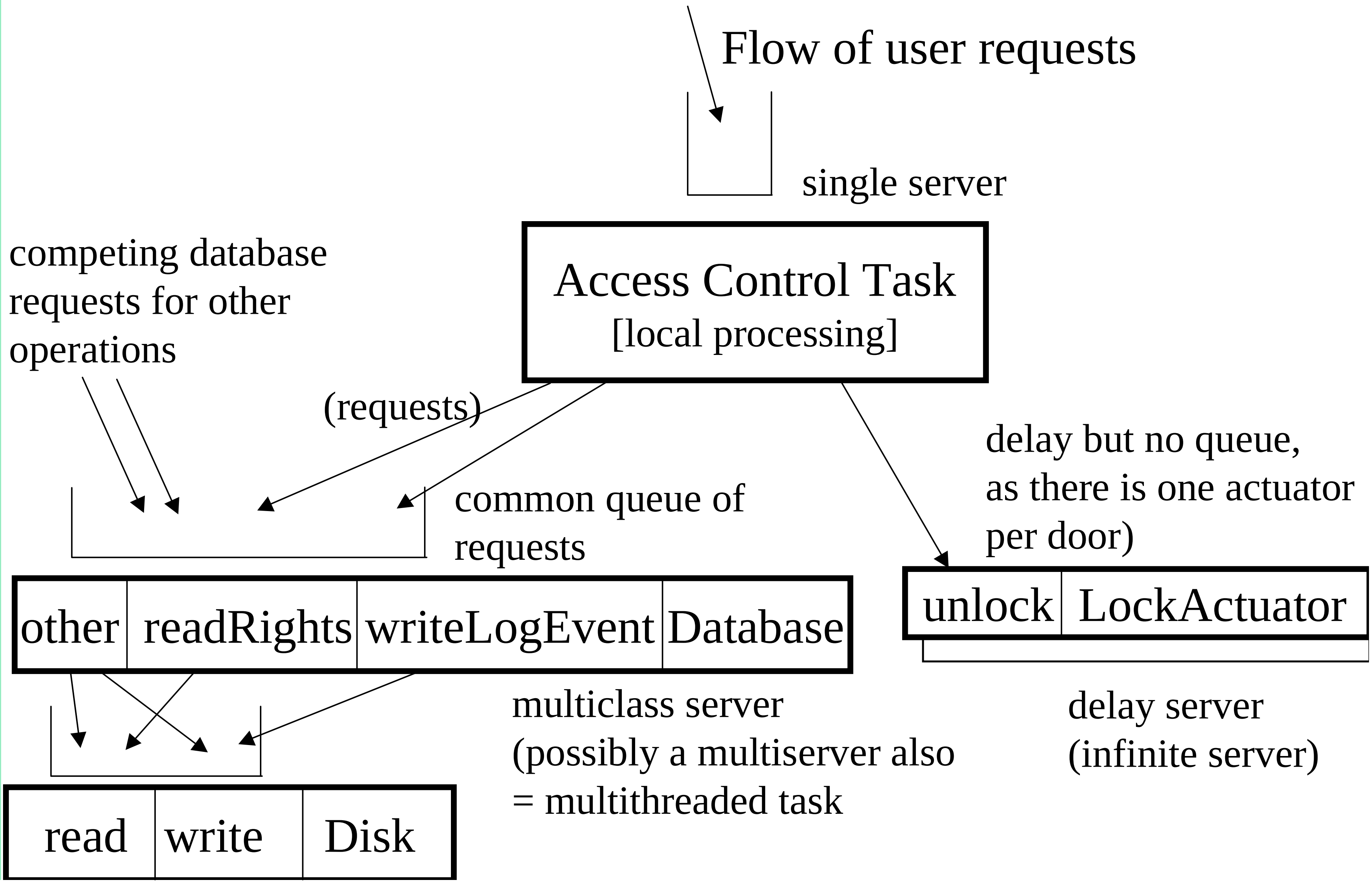
- includes lower servers
- here ***AccessControl*** includes lower layers:
  - LockActuator,
  - Database, and
  - Disk
- service time of AccessControl

  can be found recursively
- it includes waiting time at the database
- *alert* includes logging



Carleton UNIVERSITY

Layered queueing applies...



Carleton UNIVERSITY

Service time in layered queue systems...

... is not knowable without a full analysis

- because it includes contention at lower servers
- the queueing delay is affected by competing scenarios and applications

- this is the key difficulty in understanding performance in layered systems
  - for example, bottleneck location may be unstable

Carleton UNIVERSITY

Notation for layered queueing models

- by convention, call
  - all servers and resources ***"tasks"***,
  - all resource-operations ***"entries"***,
  - all requests for operations by a server ***"calls"***.
- all requests to a task enter a ***common queue***, which can have any discipline,
  - entries define ***classes*** of service
  - many kinds of tasks cannot support pre-emptive disciplines
- ***synchronous calls*** (that block the caller) are distinguished from asynchronous (that do not).
  - sync calls always lead to a single reply
  - more complex request types can be built up

Carleton UNIVERSITY

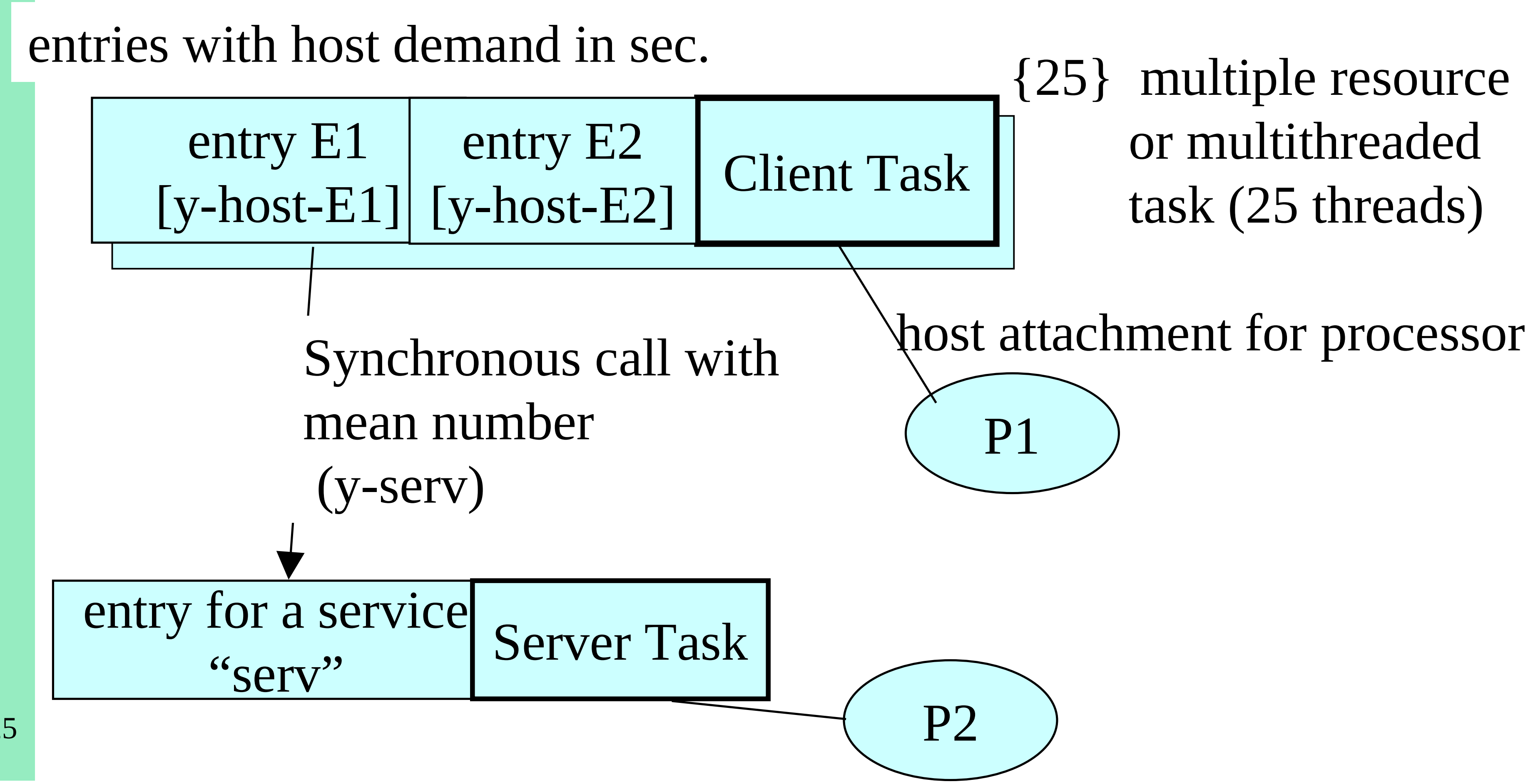
LQ Model (2)

- a sub-scenario defines ***entry behaviour***, by a sequence of operations and requests.
  - can use a default stochastic model for entry behaviour:
    - random "slices", with a given coefficient of variation
    - either random requests with given mean numbers, or deterministic numbers of requests, in random order
- there is a ***"host" processor*** server for every task, not always shown
  - host service time is divided into slices between requests for other services
  - host servers have the same semantics as tasks, all requests are synchronous, every software entry generates an entry on its host,

Carleton UNIVERSITY

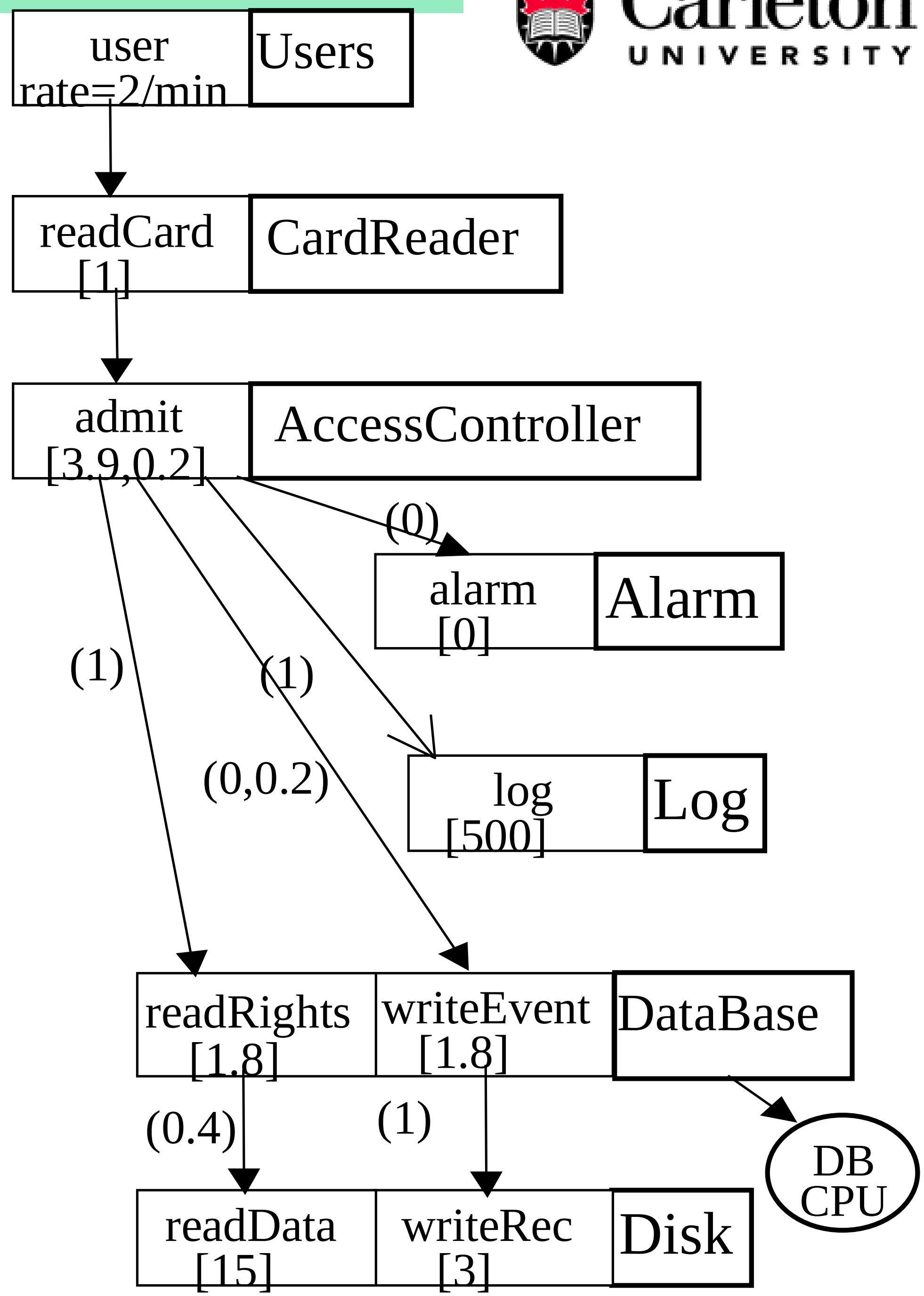
Simplified notation for a LQ model

- with default entry behaviour (random order of calls)
- parallelograms are optional....



LQN fragment for the Door Access Control

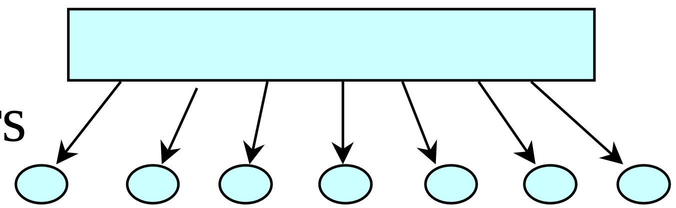
- "alarm" has no calls or load
- doesn't show database operations to store video frames
- some parameters show "second phase" operations: (0,0.2) or [3.9, 0.2]
  - after the reply



Carleton UNIVERSITY

Layered Queues are a *Canonical Generalization of Queueing Networks*

- includes ordinary QN as a single layer of client and servers
  - "program" entity calling its servers



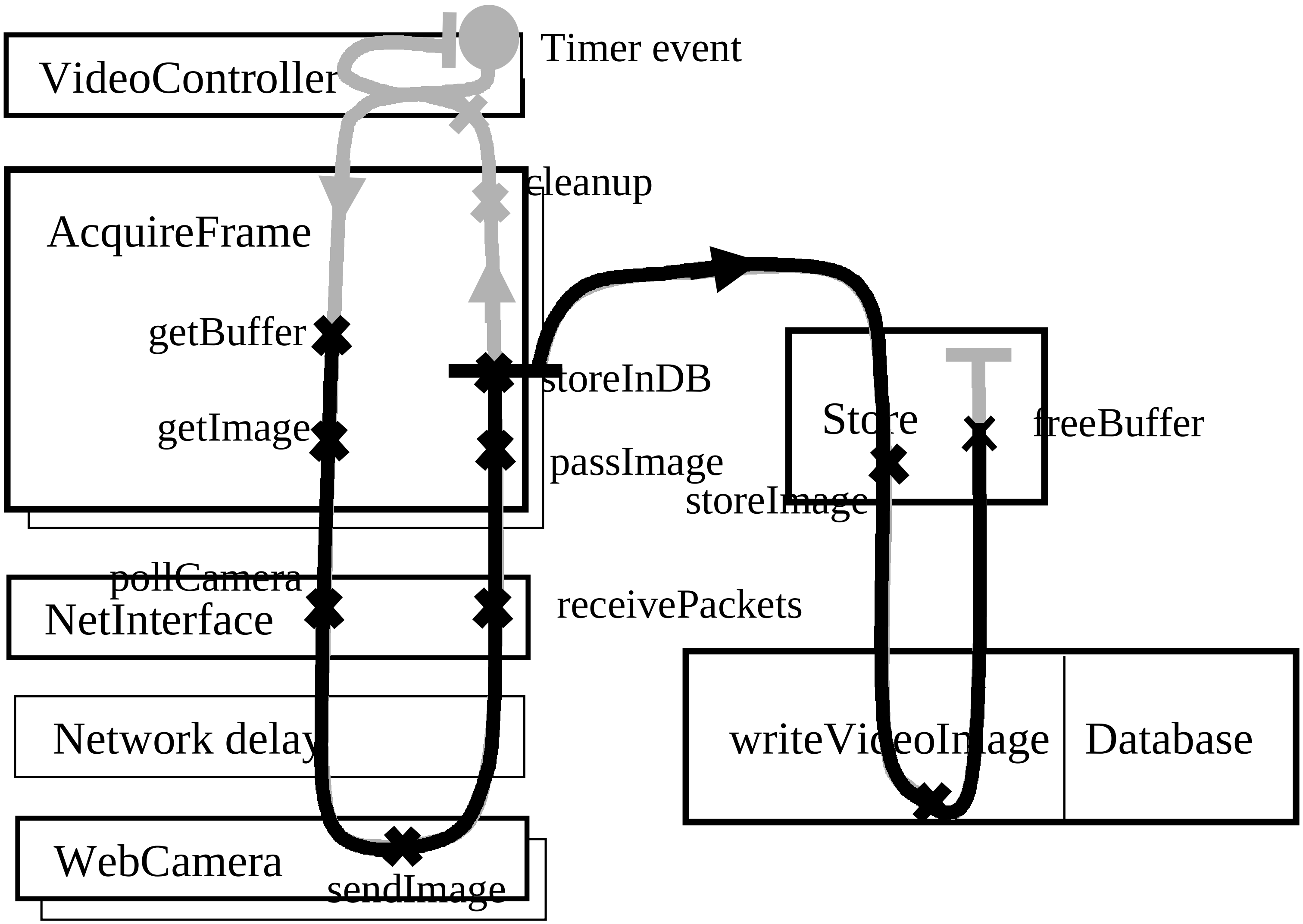
- an LQN describes any ***Extended Queueing Network*** with nested resource use
- advantage: it easily describes a system with hundreds of logical resources, many held at once, in many patterns.
- it has an ***economical, concise*** set of parameters for the stochastic default entry behaviour

Carleton UNIVERSITY

Different considerations for a logical resource: the Buffer pool for Video frames

- its service time includes parts of the execution of various tasks
  - it is not identified with any particular process
- it will be modeled by a "task" which we can call a pseudo-task
  - runs on a pseudo-host
  - no execution time of its own
  - makes calls that define the operation executed with the resource
- sometimes the place of the pseudo-task can be taken by an actual "resource manager" task, if it exists

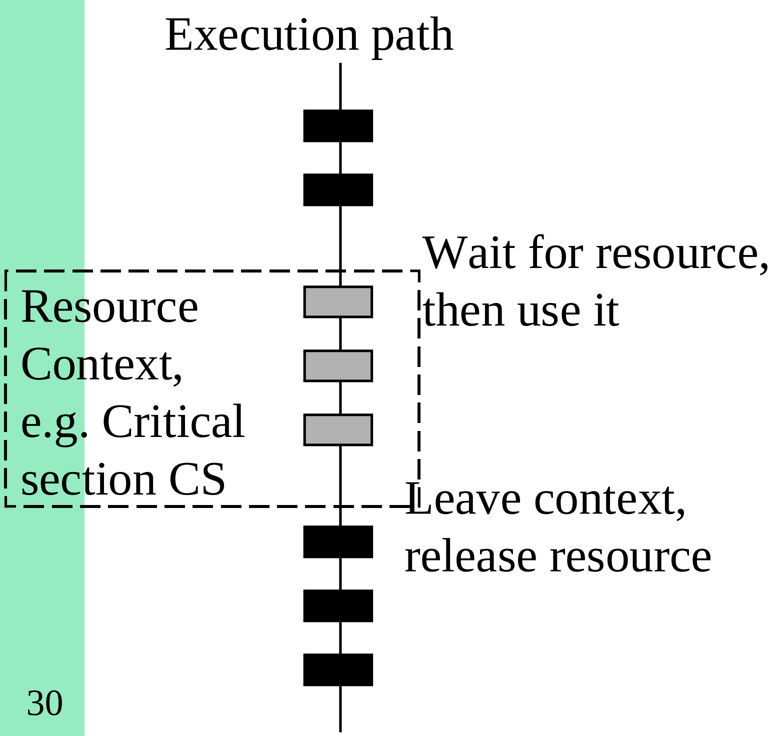
Path showing the holding time of a buffer

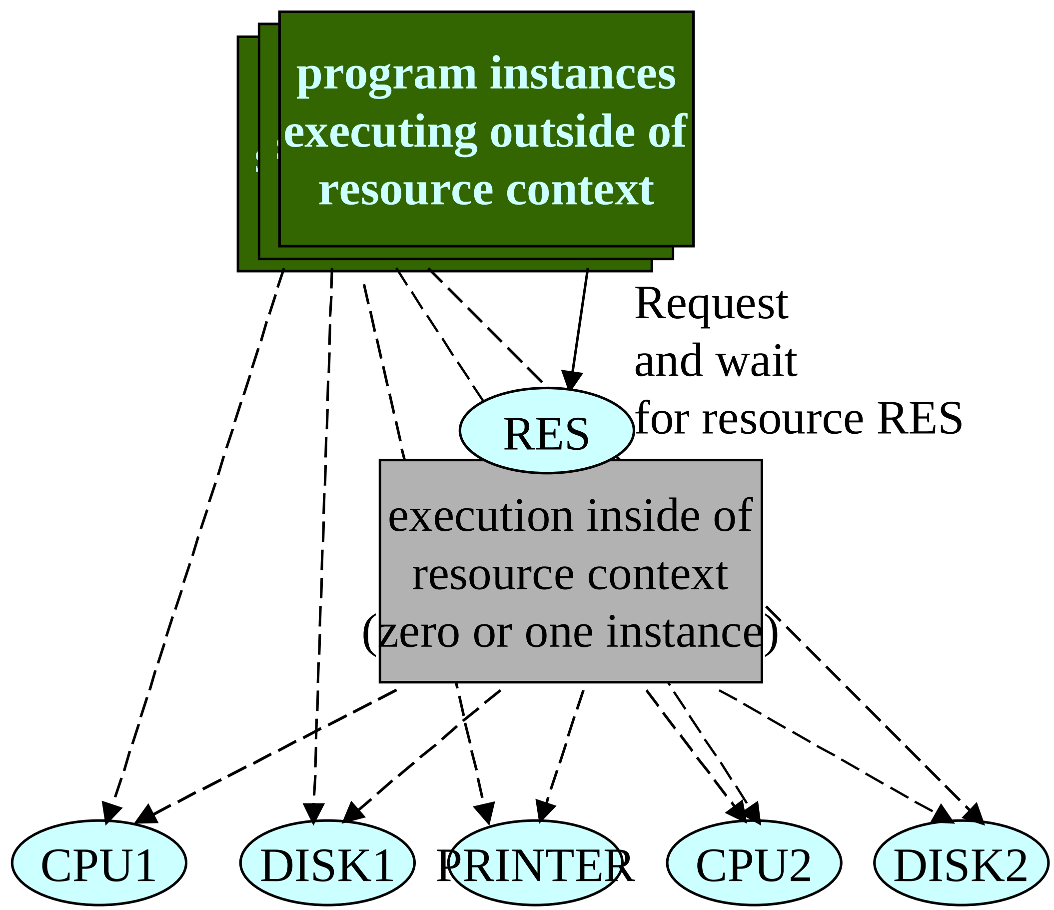


Carleton UNIVERSITY

Modeling technique for a logical resource

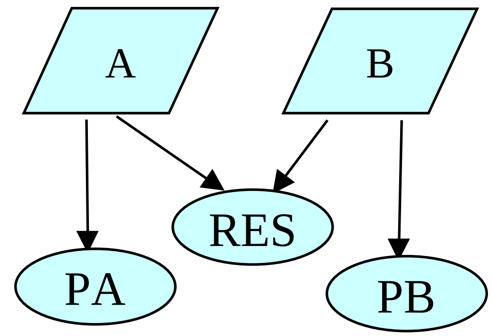
- A buffer or critical section ….





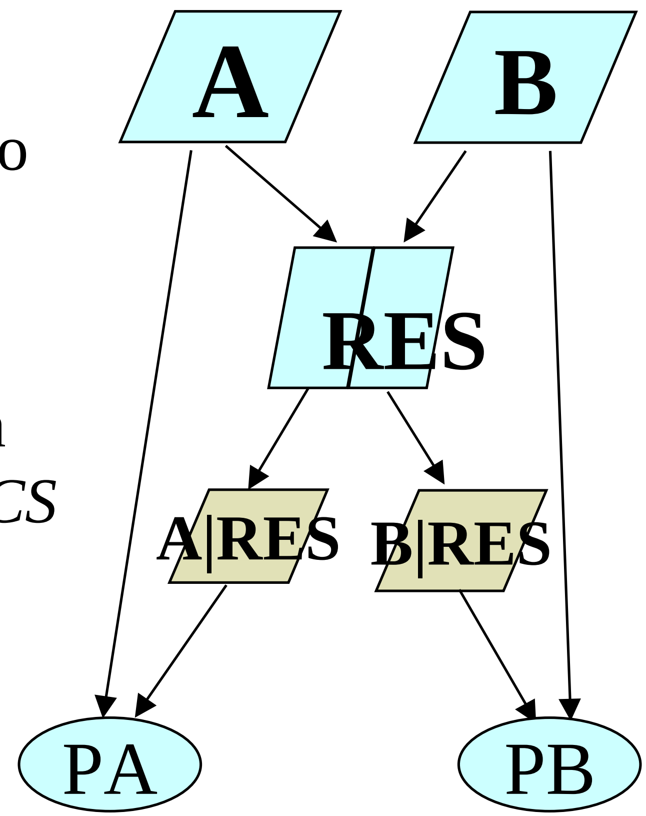
Carleton UNIVERSITY

Resource pseudo-task when user tasks are distributed...



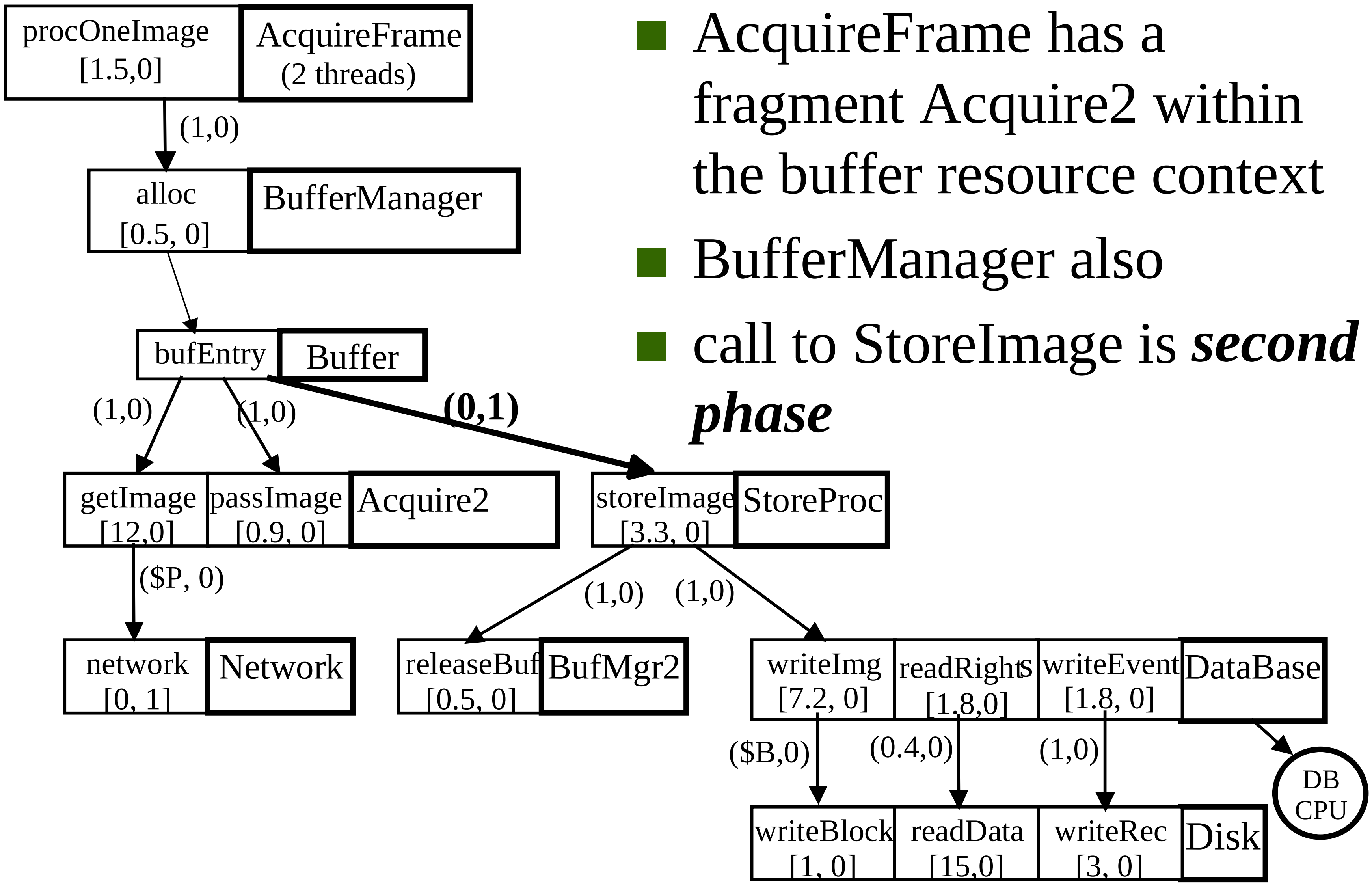
not a suitable model

- Tasks A and B must enter a critical section (call it CS) for some work....
  - but this doesn't express what they do within CS
- So:
  - Separate out the computation within CS into *Shadow Tasks A|CS* and *B|CS*
  - to direct the call from A to A|CS, make CS a *pseudo-task* with two *pseudo-entries*



Carleton UNIVERSITY

LQN fragment for the video buffer

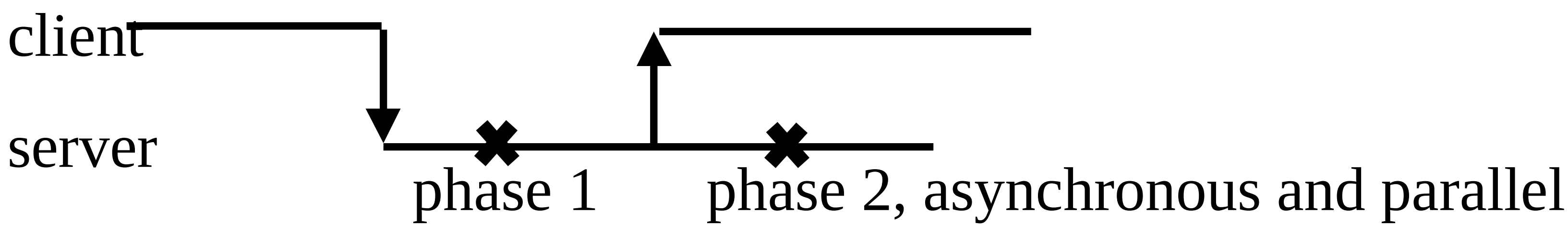


- AcquireFrame has a fragment Acquire2 within the buffer resource context
- BufferManager also
- call to StoreImage is ***second phase***

Carleton UNIVERSITY

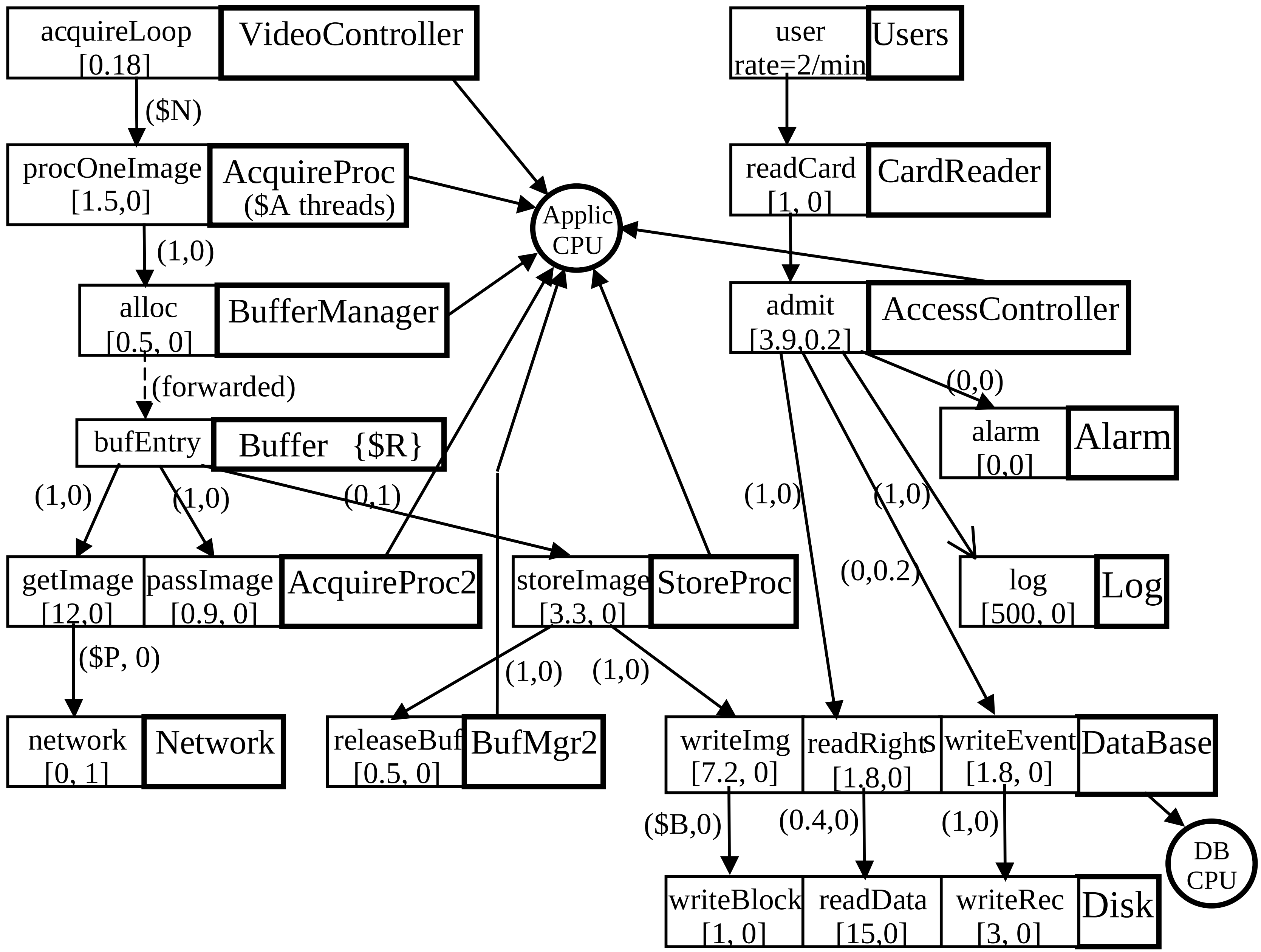
"Second Phase Service" in software servers

- Idea: often used to enhance performance
  - give a reply as early as possible
  - Do postponeable work after the reply, as "phase 2"

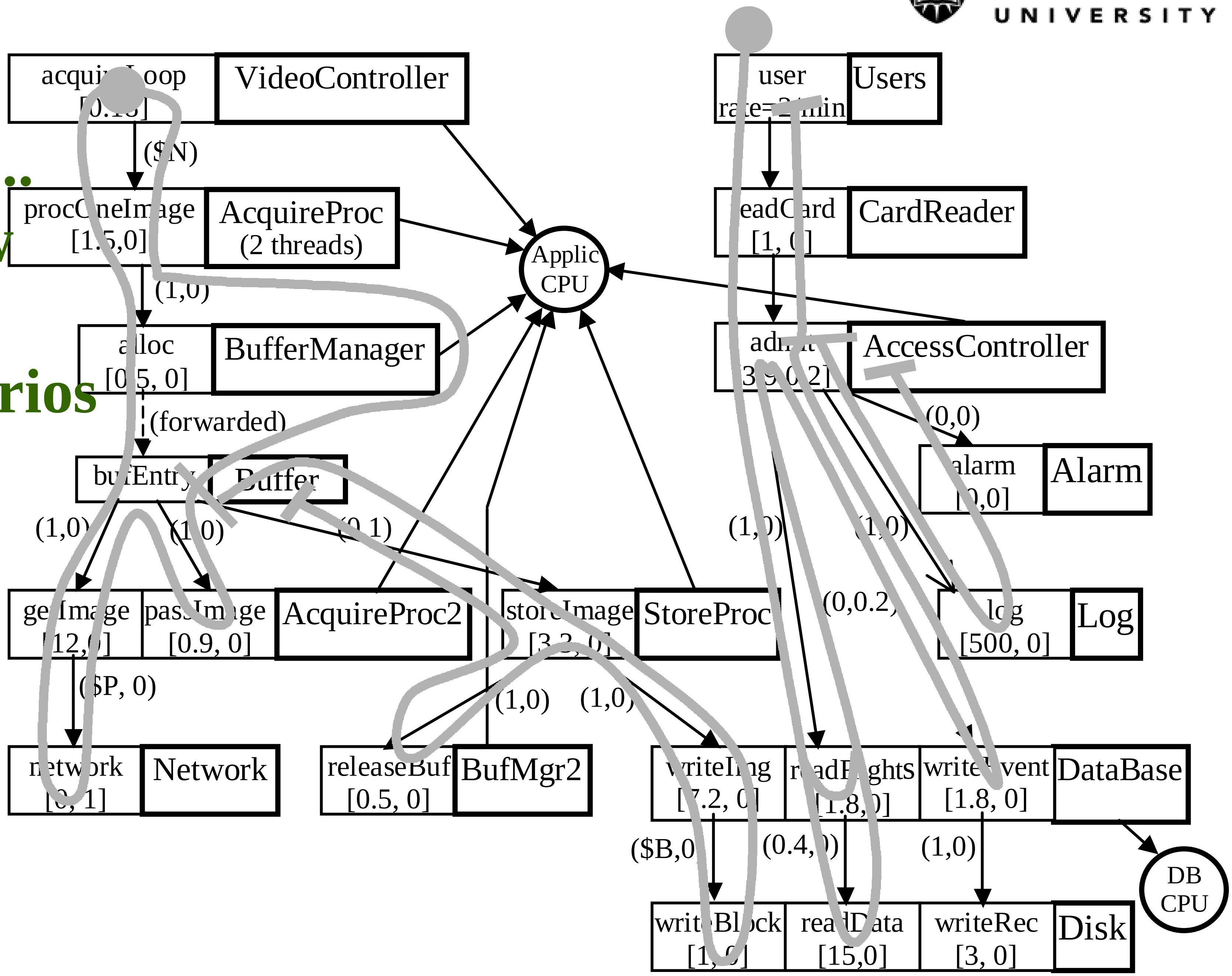


- e.g..: Database server update operation:
  - write to log file before returning, execute final writes later.
- Second-phase model may
  - place this work right after the return (approx), or
  - send a message to a clean-up process that does it later
- Queueing approximation paper in Performance 99

Overall LQN: $N cameras, $R buffers



BSS LQN... follow the scenarios

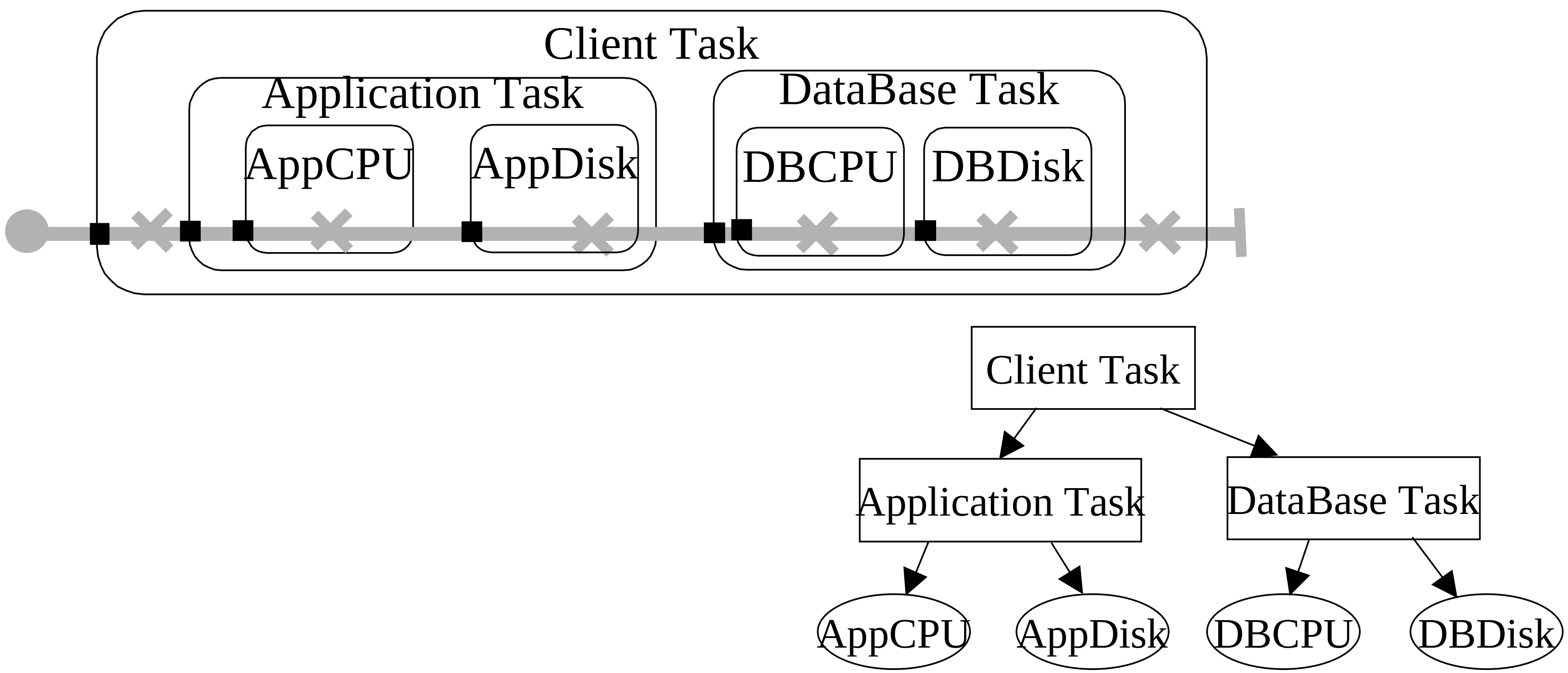


Results for varying buffers.

Buffers	Cameras	Prob(miss)
1	10	0
1	20	0.001
1	30	0.417
3	10	0
3	20	0
3	30	0.003
3	40	0.319

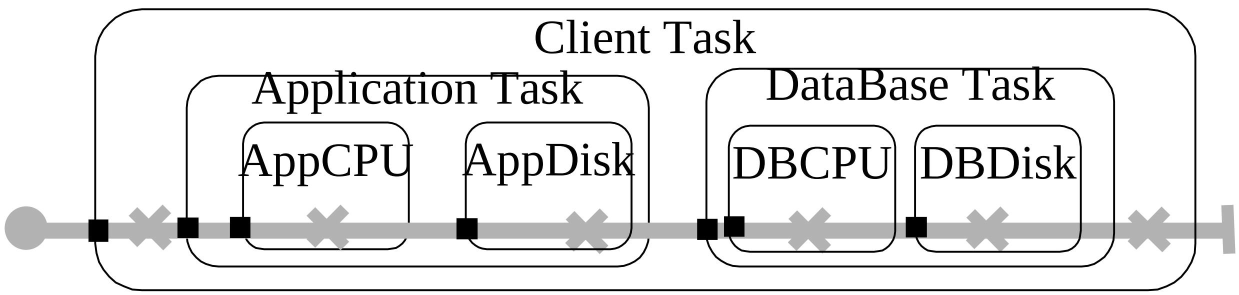
Building layered models

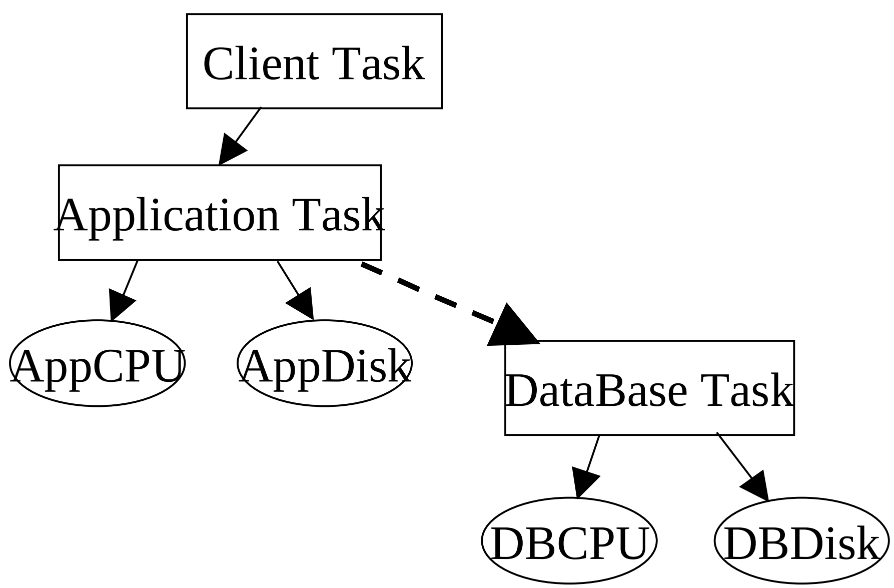
- identify resource contexts
- identify interaction patterns between them
  - synchronous, asynchronous, forwarding
- synchronous: request-reply from nested resources



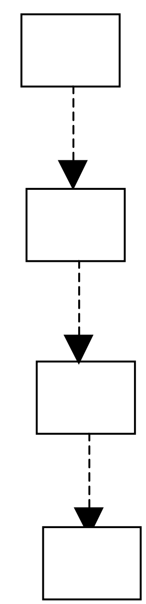
Building layered models (2)

- ***forwarding*** shows execution passed directly from one server to another, without returning in between
  - may traverse a route before returning



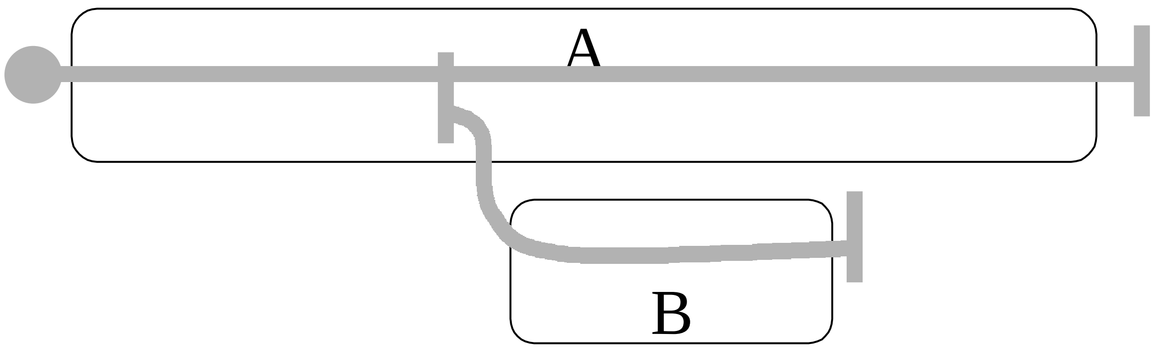


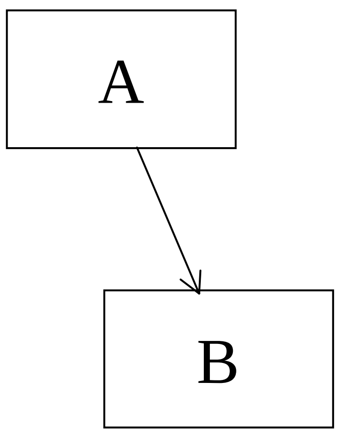
4 in sequence:



Building layered models (3)

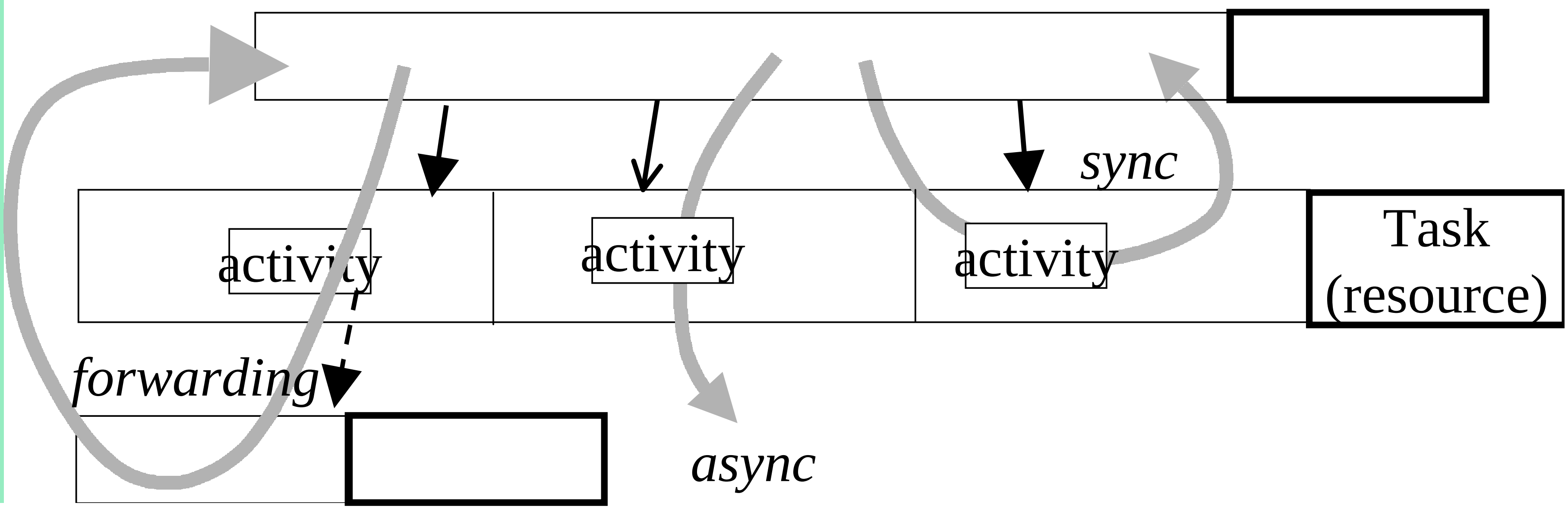
- forking to a new context is an ***asynchronous*** interaction





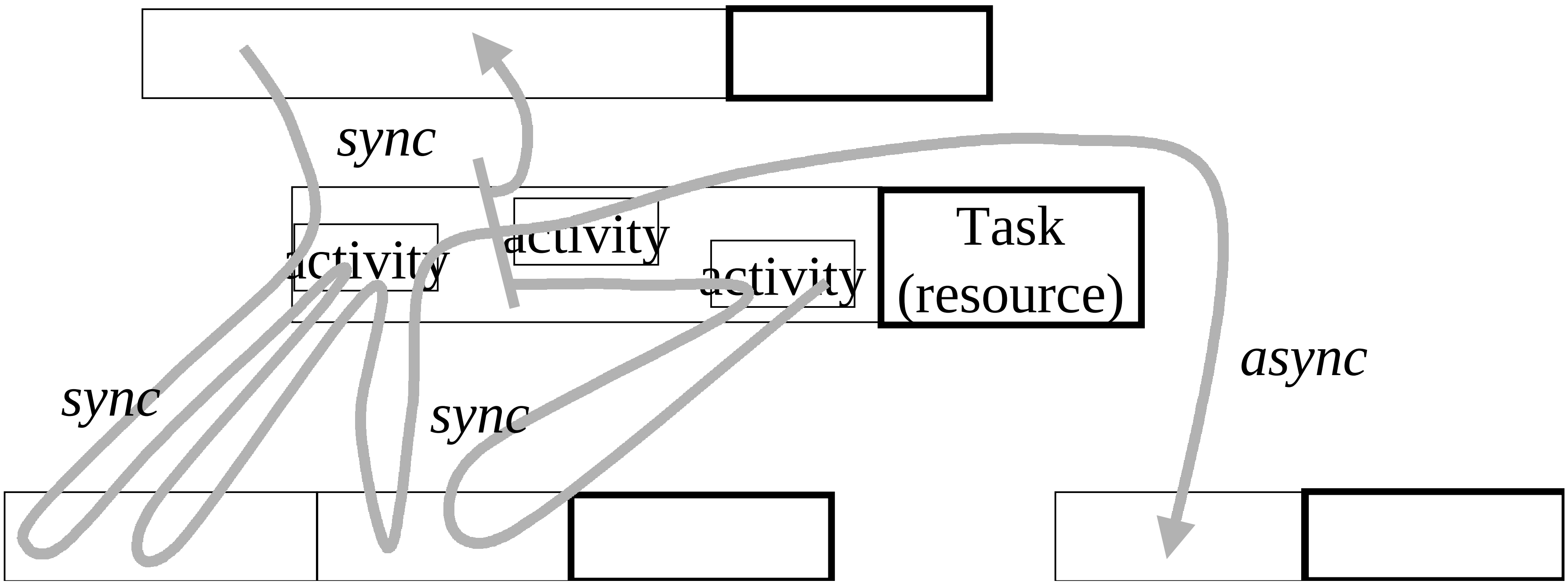
Interaction summary

- Resource == "task"
- Resource-operation (simple): one activity....
- requests for other resources:
  - expressed via interaction types: sync, async, forward

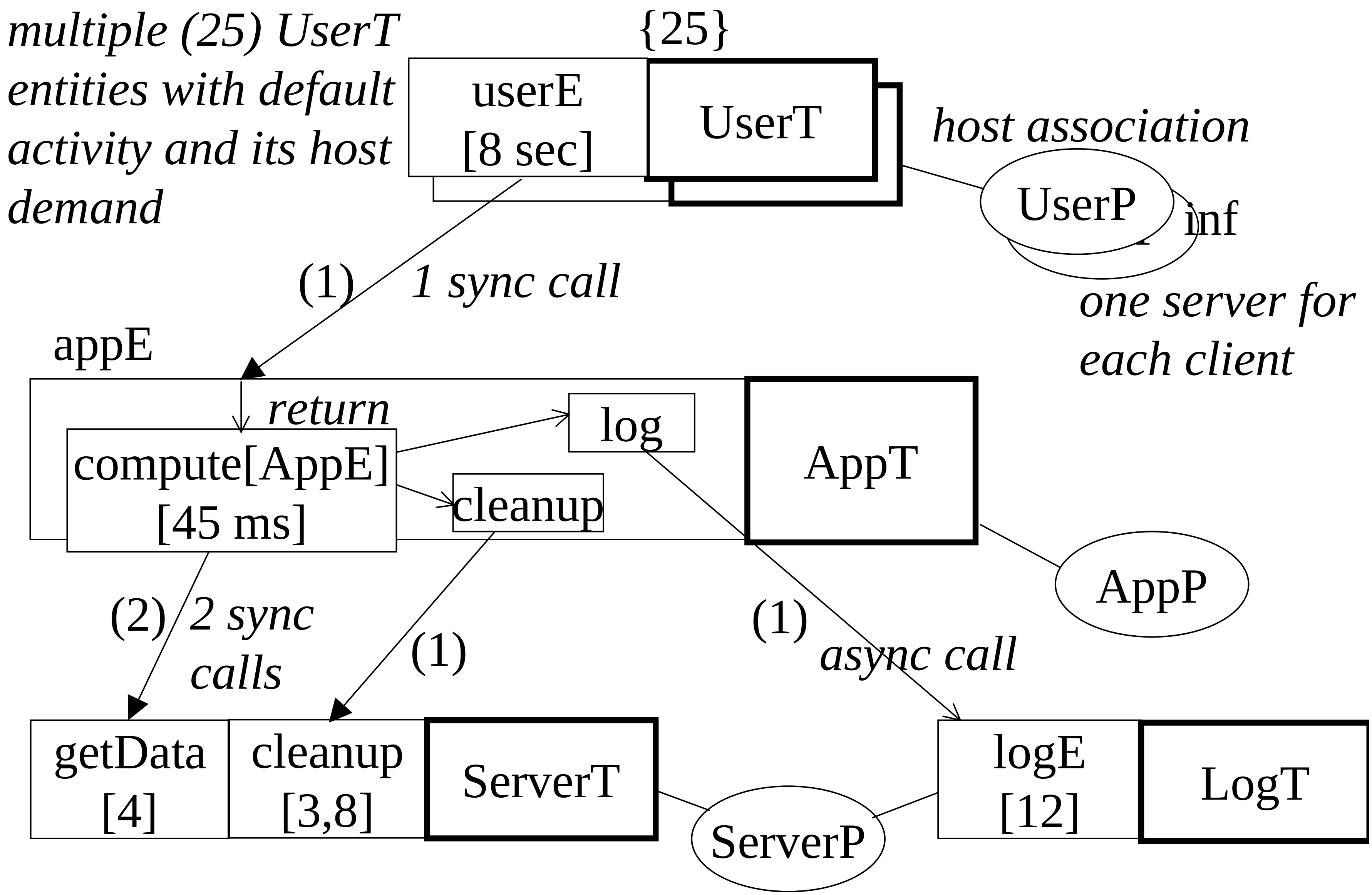


Activity sequence detail in an entry

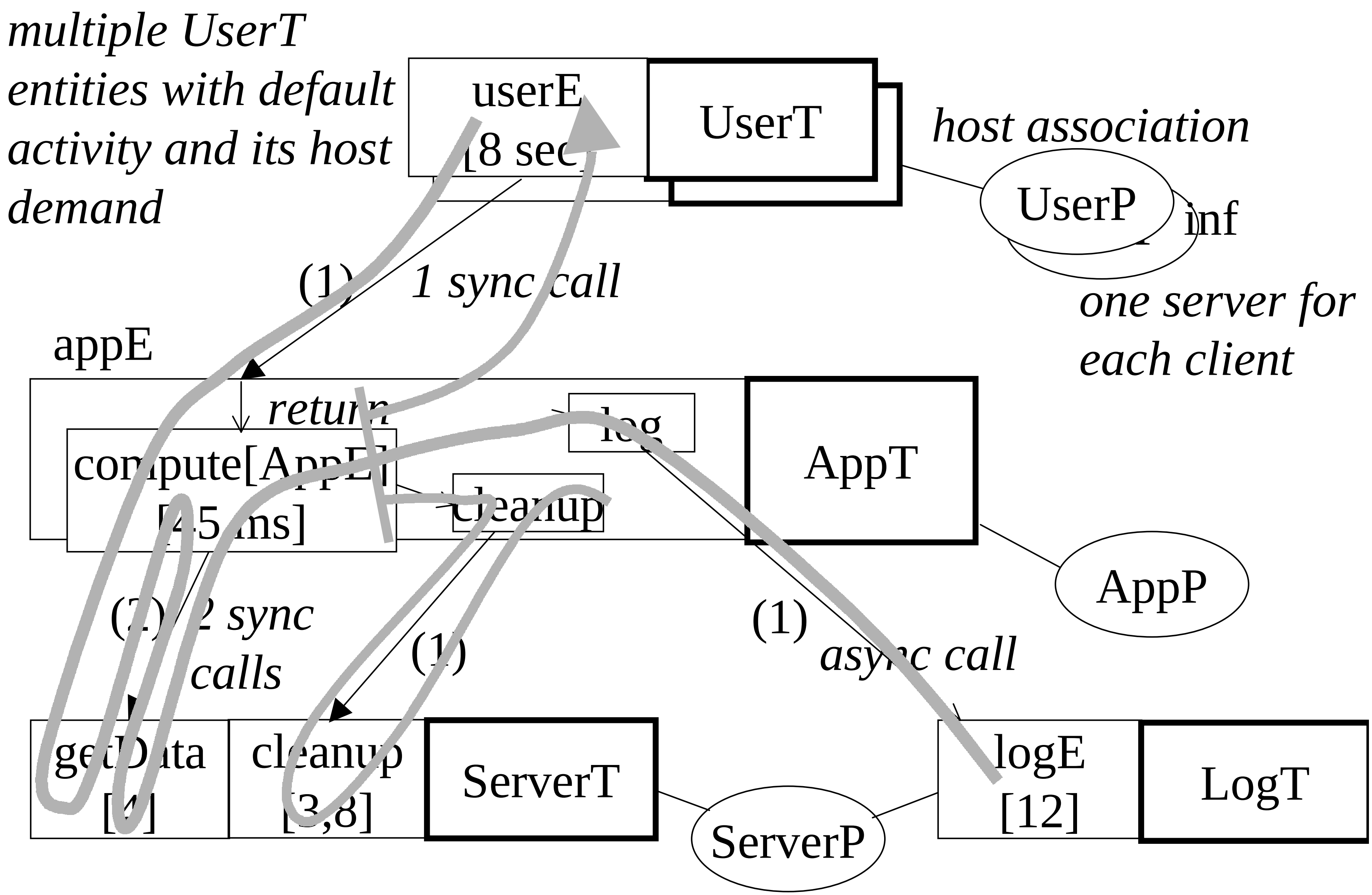
- an activity has a workload (CPU demand and requests)
- sequence relationships
  - (AND/OR fork/joins)
- interaction types again: sync, async, forward



An example with activity sequence detail

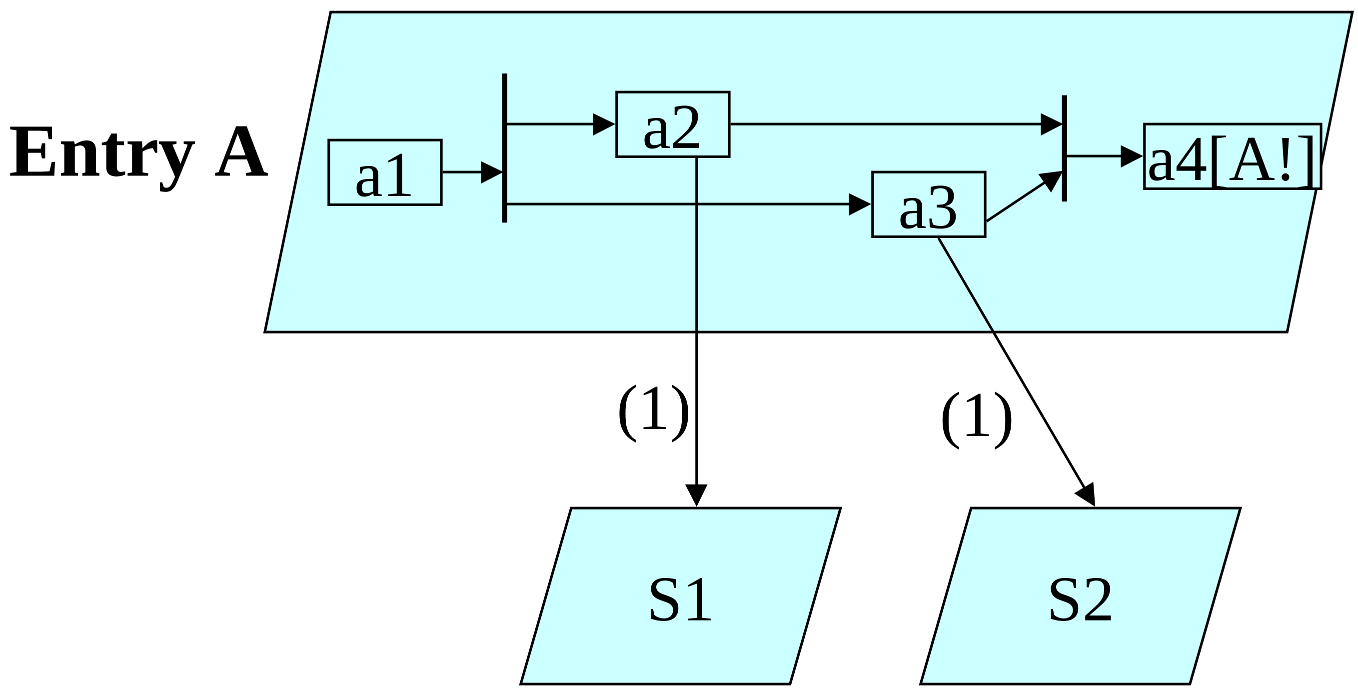


Example: visualize the scenario



LQN pattern for parallel operations

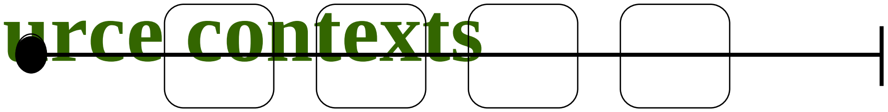
- Server A requests services from S1 and S2 in parallel, that is it sends both requests and then waits for both replies.
- This happens once during an execution of an entry A:



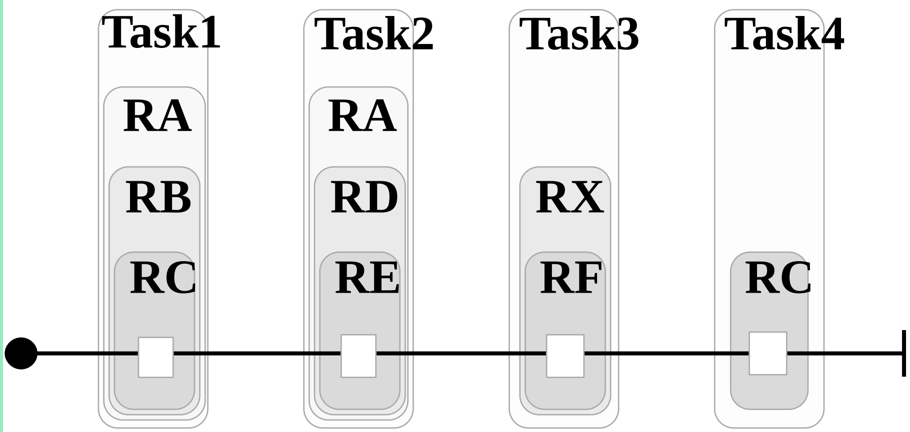
Carleton UNIVERSITY

Some resource context patterns...

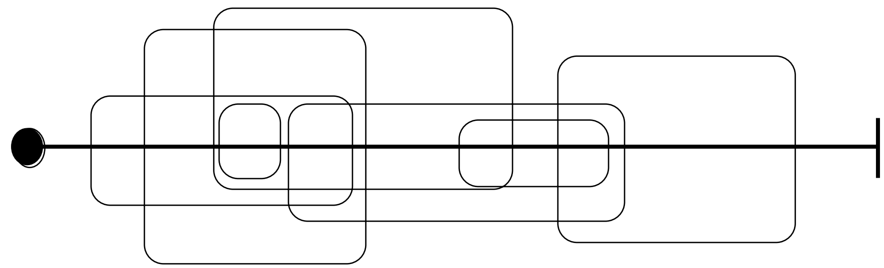
Other patterns of resource contexts



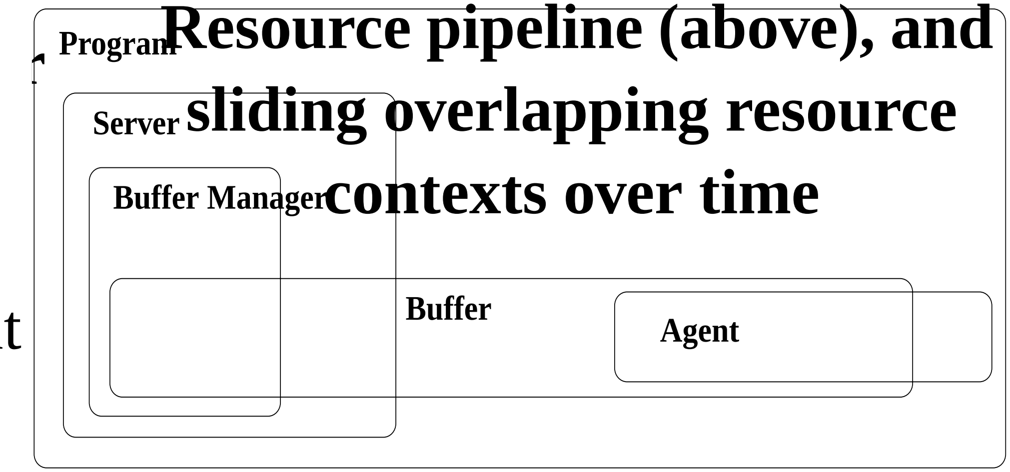
Pipeline



A separate context for each activity



Chaotic, unstructured



Resource pipeline (above), and sliding overlapping resource contexts over time

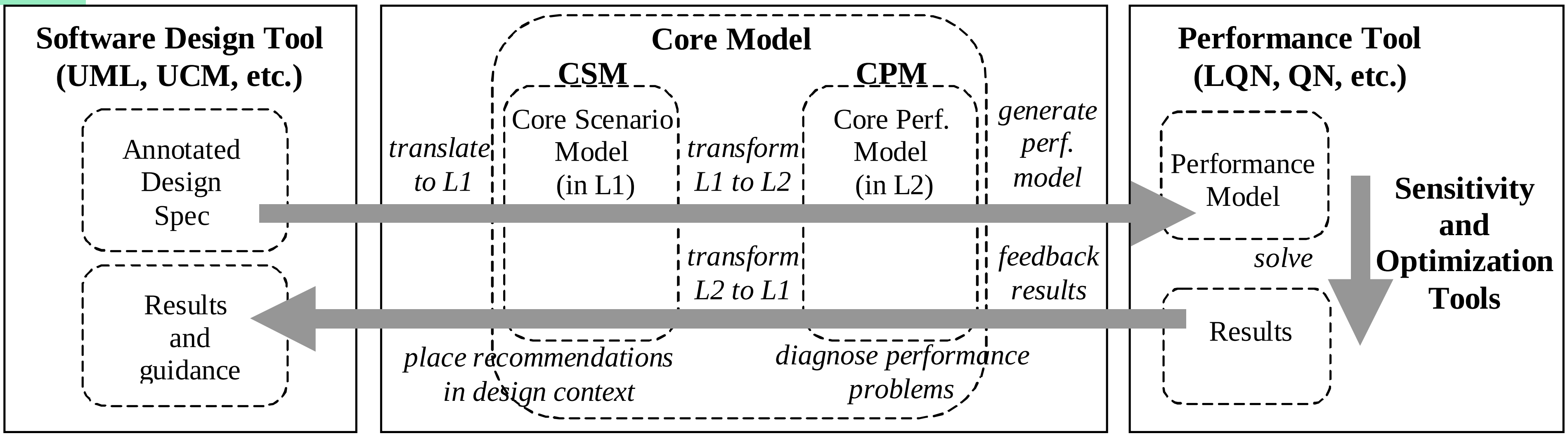
Resource pass-back:
... an interesting pattern that needs work...... the Buffer is released by the Agent

Carleton UNIVERSITY

Summary of model-building from software descriptions

- can be seen as a generalization of the methods defined by Connie Smith
- based on tracing scenarios, detecting resources and interpreting nesting and interaction types
  - from scenarios in ***Use Case Maps***: TOOLS 2002 paper
  - from tracing (***"angio traces"***): MASCOTS 95 and TSE 2000 (Hrischuk)
    - ("TLC" = trace-based load characterization)
  - now analyzing ***UML scenarios***, expressed with the UML Profile on Schedulability, Performance and Time (2002)

Proposed PUMA toolset architecture......
(Performance by Unified Model Analysis)



- *general* software model input via CSM (not only UML)
- *general* performance model types via CPM (not only layered queues)
- includes heavy element of model investigation, sensitivity tools, optimization
- proposal also for component libraries for completions

Carleton UNIVERSITY

LQ network *solvers*

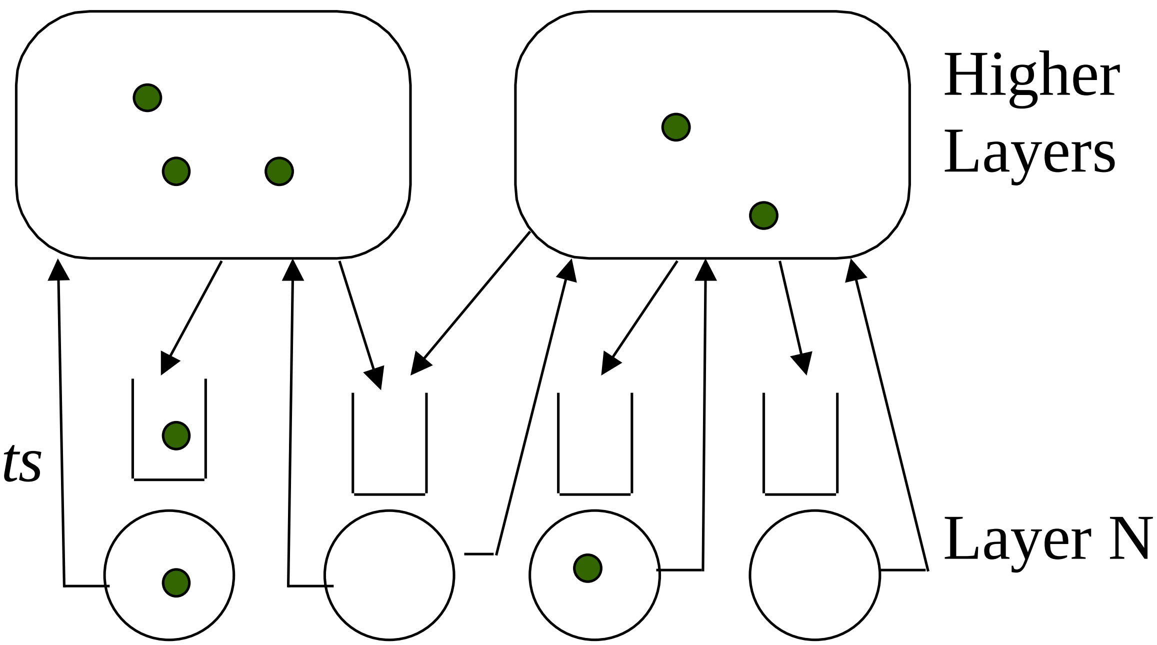
- ***www.layeredqueues.org*** site to provide resources
- LQNS (Franks/Rolia/Petriu/Sevcik/Woodside) (84 on)
  - iterative basic MVA with lots of approximations for variance, multiservers, parallel paths, and other aspects
- Fontenot described one open layered server (Sigmetrics 1988)
- Petriu (1993) Markov model by TDA "Task Derived Aggregation"
- Ramesh/Perros (98 - 2000)
  - open systems, with close attention paid to variance effects, and a structured sequence of classes of service
- Kahkipuro (UML2000), a basic multilayer solution
- Menasce (2 layers) (2002) for critical sections, etc.

LQNS: Iterative MVA Solver for LQN

"Active Server" closed queueing sub-model for each layer (1984)

"Delay Server" with tokens that represent clients or active servers from layers above

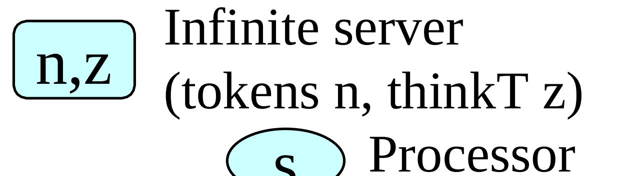
Tokens represent requests from upper layer servers

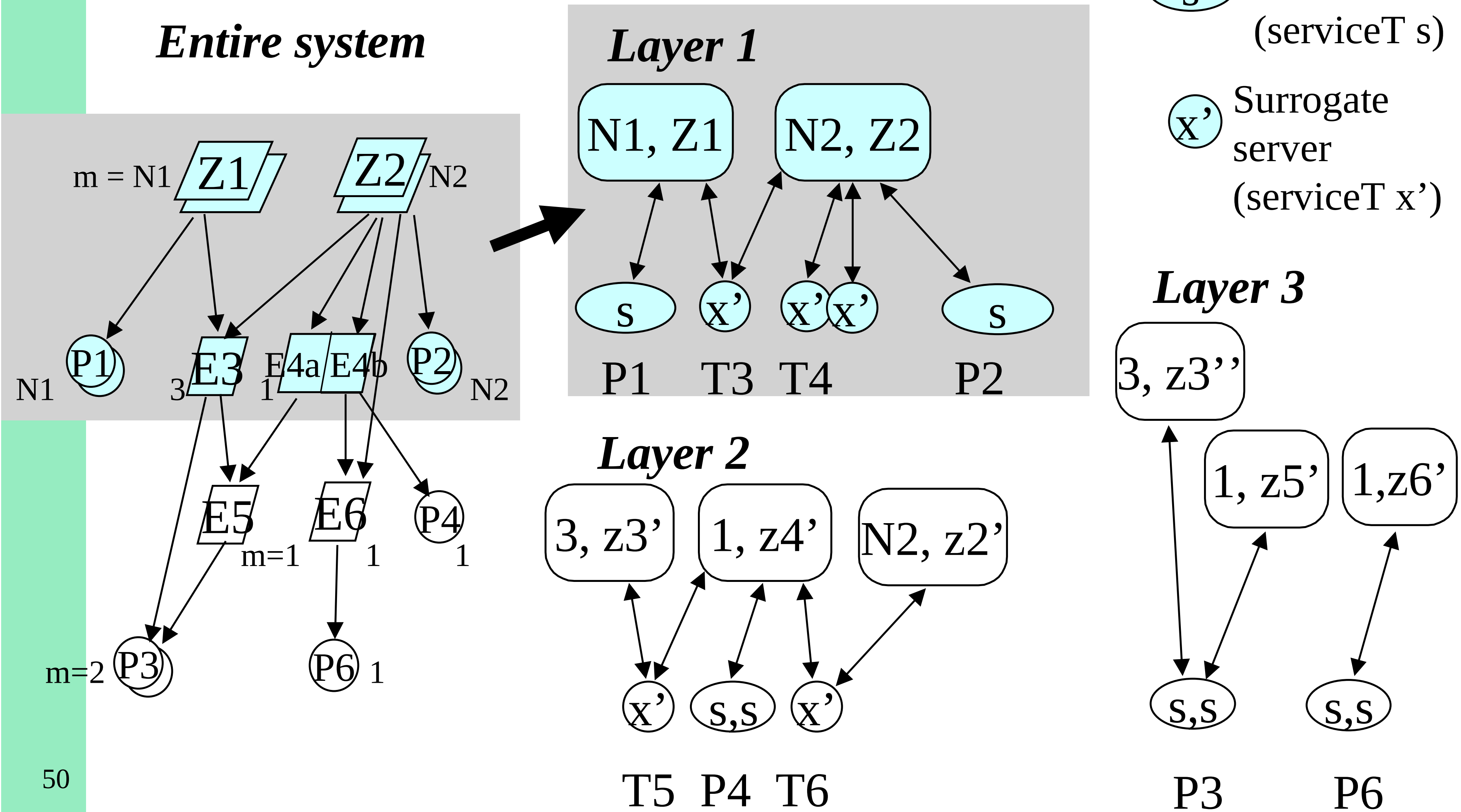


Servers represent server tasks at layer N:

- *Service times of the servers include delays at lower layers, including processors*
- *Servers may be non-standard (two phase!)*

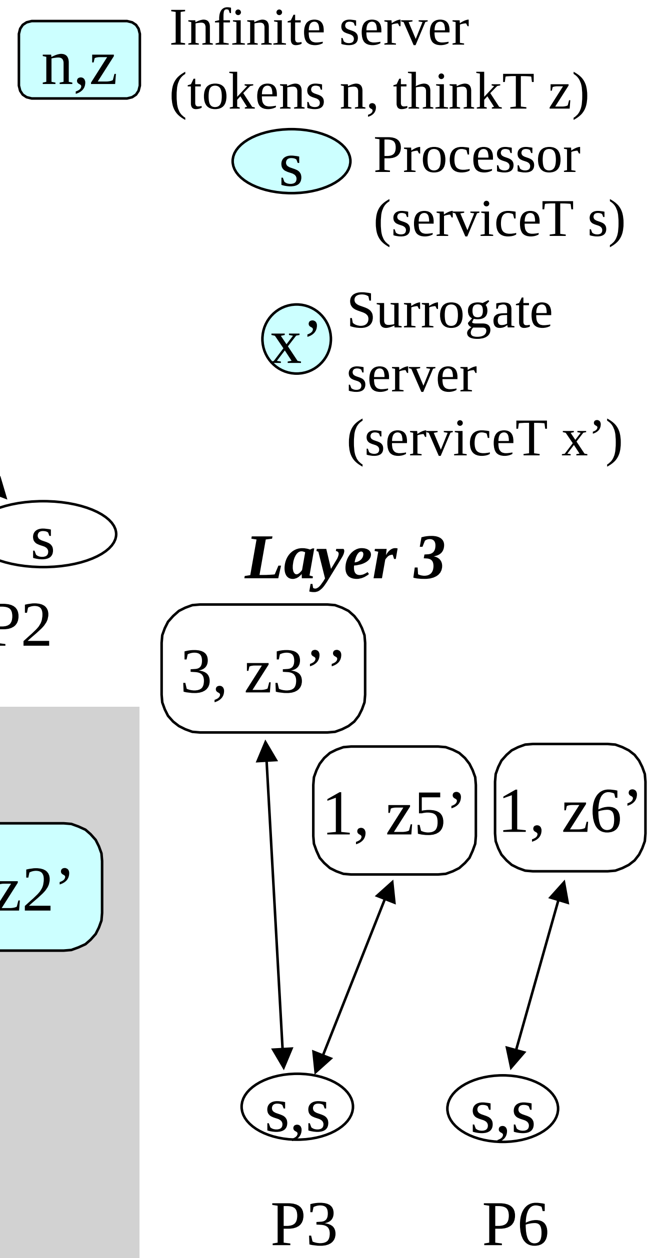
"Layerize": Layer 1 Submodel

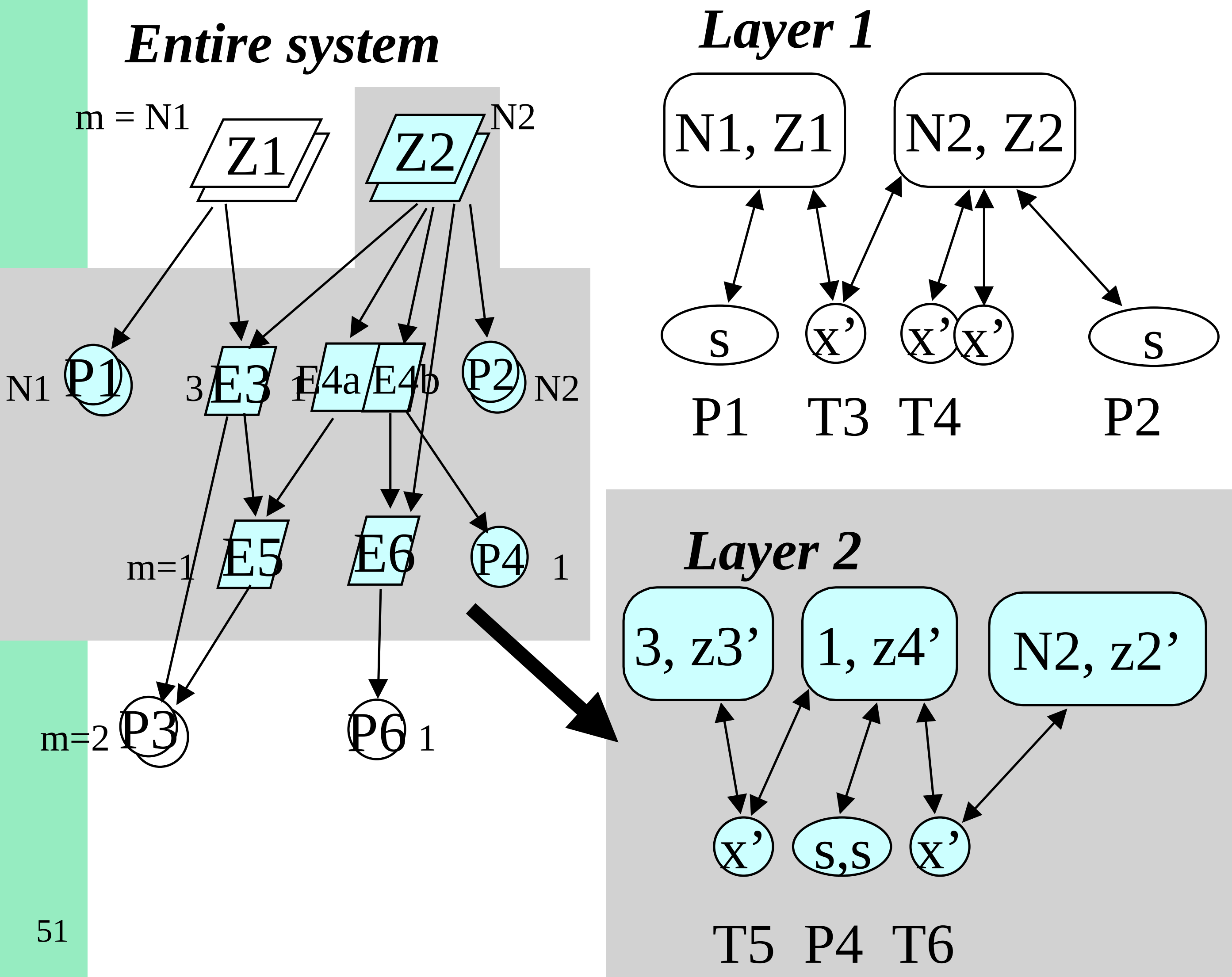




Carleton UNIVERSITY

"Layerize": Layer 2 Submodel

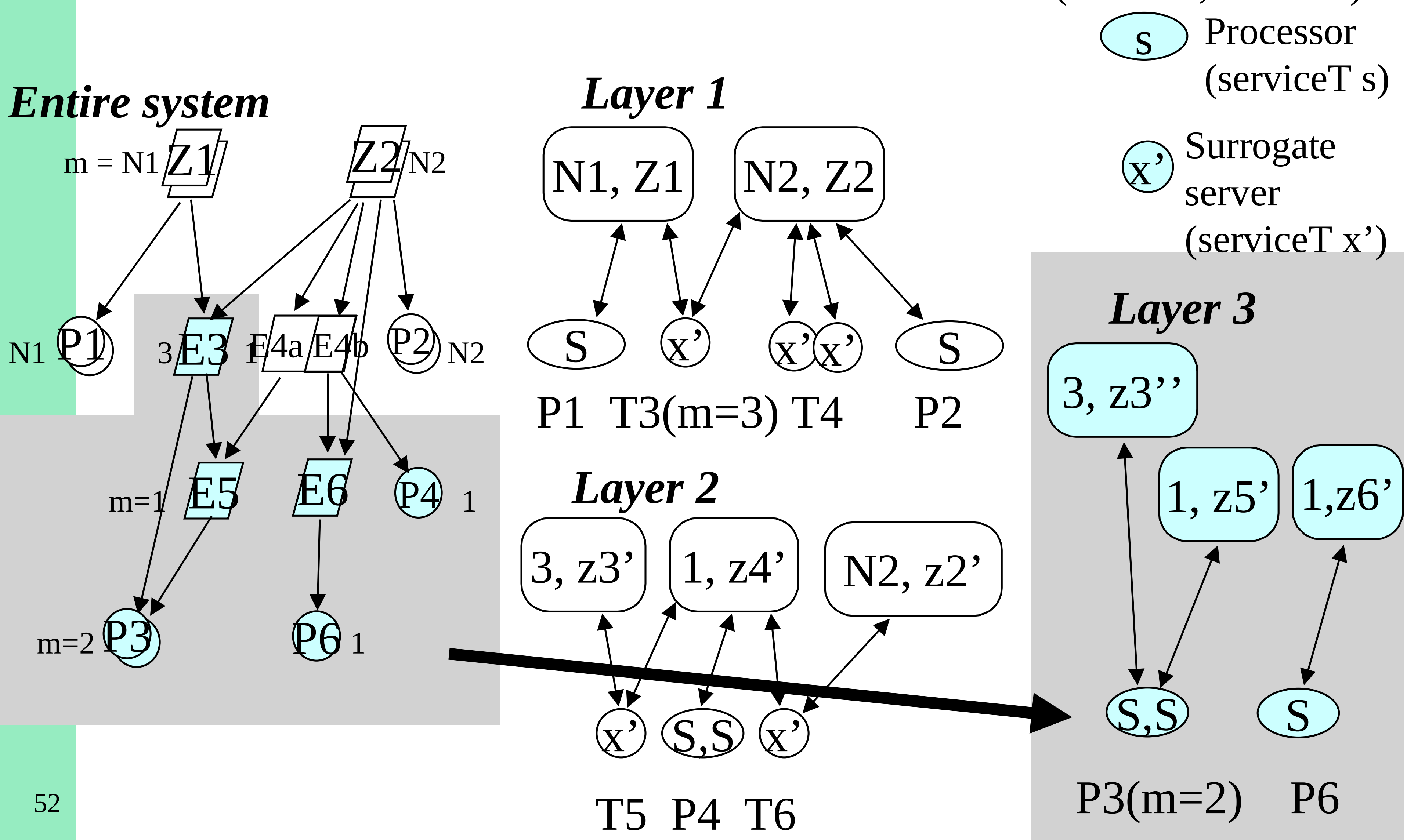




Carleton UNIVERSITY

"Layerize": Layer 3 Submodel

n,z Infinite server (tokens n, thinkT z)



Carleton UNIVERSITY

Layerizing strategies can be interesting

- Woodside 1984/ 88/ 95...one task per layer
  - calls can jump layers (causes a dependence effect "interlocking"
- Rolia/Sevcik 1988/92/95... wide layers, greedy from the bottom.
  - strict layering (calls cannot jump over)
  - accommodation for jumping over
- Ramesh/Perros 1998... strict layering
- Franks thesis 1999.... in LQNS... flexible choice, OR greedy from the top, OR all in one big layer (!!)
  - balanced layer sizes are best for solution time.
  - a detailed submodel for dependence effect
  - detailed study of second phases

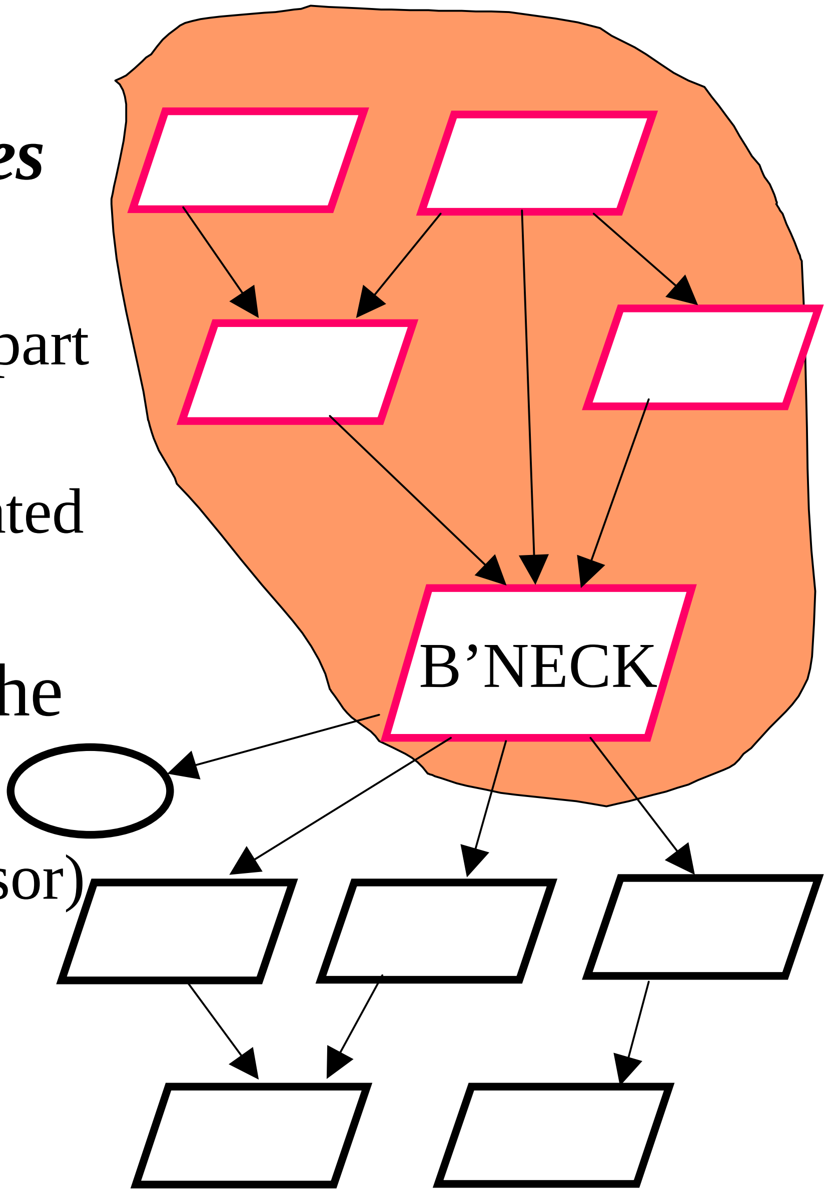
Exploiting the solutions to improve the software design

- sensitive parameters point to aspects with most leverage
  - sensitive execution demands can be reduced
  - sensitive interaction counts can be reduced
  - long entry service times can perhaps be parallelized
- sensitivity is highest around bottlenecks!!
  - look at resource utilization
    - task utilization includes all its nested service times when it is blocked
    - reduce service time of bottleneck server
  - increase resource multiplicity (buffers, threads)

Carleton UNIVERSITY

Recognizing layered bottlenecks

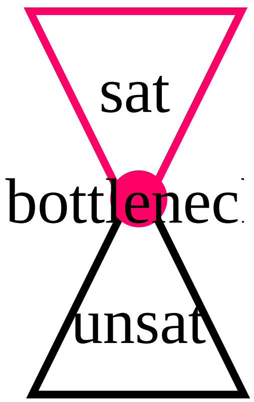
- a saturated server
- but.... a saturated server ***pushes back*** on its clients
  - the long waiting time becomes part of the client service time!!
  - result is often a cluster of saturated tasks above the bottleneck
- thus: the "real" bottleneck is the ***"lowest"*** saturated task
  - its servers (including its processor) are not saturated
  - some or all of its clients are saturated



Recognizing the "real" bottleneck

- a saturated task with unsaturated servers and host
- ***Strength*** measure $(U_B/M_B)/[\max (U_S/M_S)]$
- IEEE TSE paper 1995

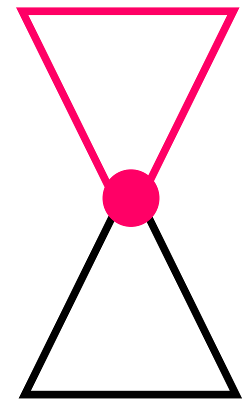
- ***Notice that:***
  - if the bottleneck task has no servers, its host utilization is the same as the task (it only computes)
  - so it must have at least one additional server, a device (e.g. disk), task, or other logical server
  - also, it must have sufficient clients to build a queue
- thus, there is often an hourglass pattern



Carleton UNIVERSITY

Bottleneck patterns and *threads* or multiplicity

- a task with M threads counts as M concurrent servers or clients
  - in identifying the "hourglass" pattern
- in the "strength" measure, a server with M threads saturates at U = M
- a (very rough) rule of thumb for threads, based on potential needs for concurrency:



$$M = \text{min of } \begin{cases} \text{(sum of server threads, +1)} \\ \text{(sum of client threads)} \end{cases}$$

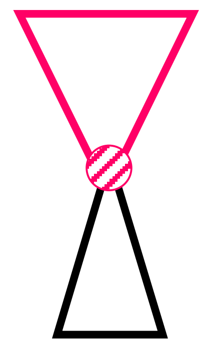
Curing a bottleneck

(1) provide additional resources at the bottleneck

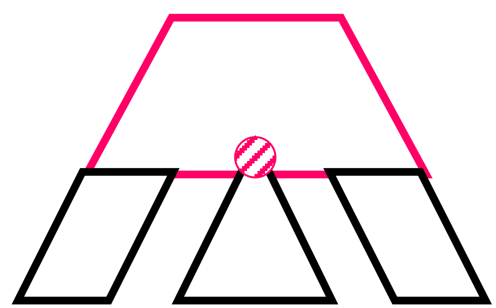
- for a software server, provide ***multiple threads***
  - some "asynchronous server" designs provide unlimited threads
- replicated servers can split the load and distribute it
- for a processor, a ***multiprocessor*** (or faster CPU)

(2) reduce its service time:

  - reduced host demand
  - reduced requests to its servers

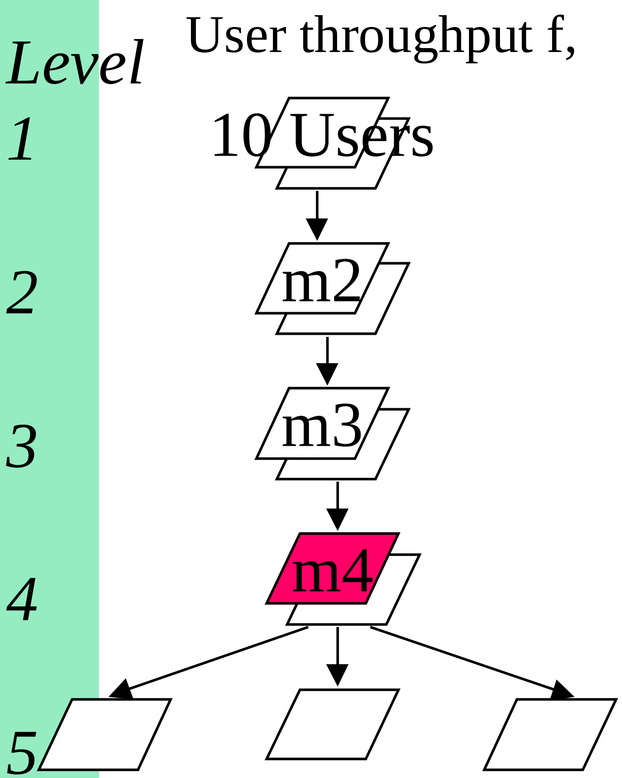


(3) divert load away from it



Carleton UNIVERSITY

Curing Software bottlenecks by multithreading



(single servers at the bottom)

- bottleneck at task 4 limits the user throughput f
- f depends on the threads at all servers
  - m2 threads for task2, m3 for task 3, etc

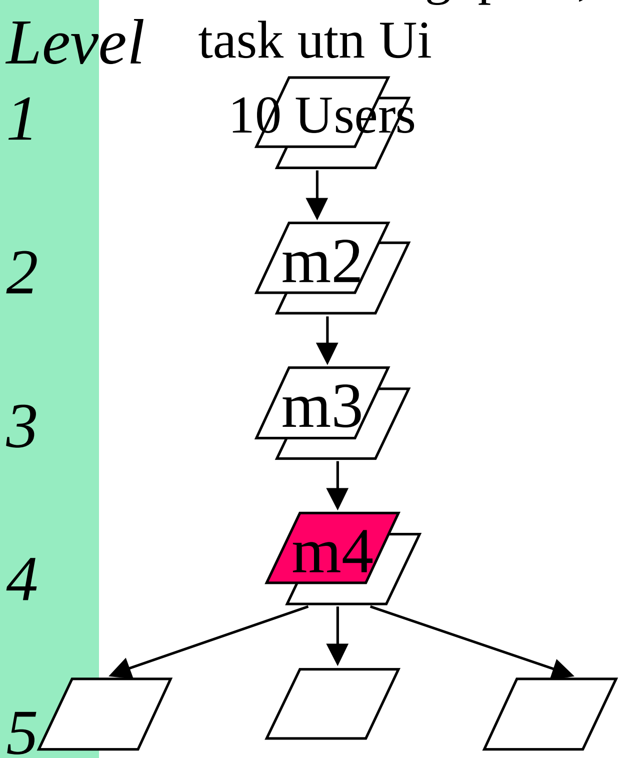
...a multi-threaded server behaves like a multi-server; two threads can execute in parallel. *If they are sequentialized by their processor servers, that appears as waiting*

- 1 sec host demand at each server, one request to each lower task
- Ui = task utilization at level i

(m2, m3, m4)...	f	(U2,	U3,	U4,	U5)
(1, 1, 1).....	0.166,	(1.	0.83	0.67	0.167)
(2, 1, 1).....	0.200,	(0.96	1.	0.8	0.2)
(3, 2, 1)....	0.223,	(2.9	1.64	0.89	0.22)
(6, 5, 4)....	0.475,	(5.5	3.9	2.75	0.475)
(10,10,10)..	0.65,	(9.3	7.8	6.2	0.65)

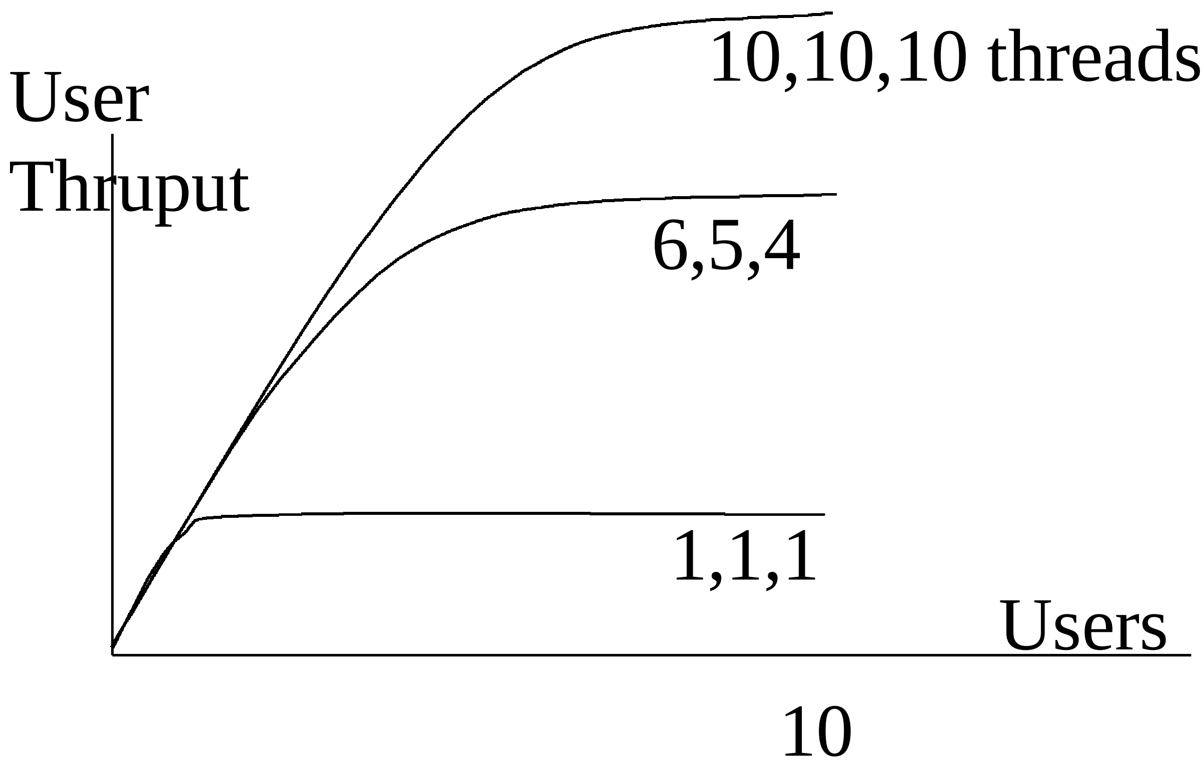
Software bottleneck relief by multithreading

User throughput f,
task utn Ui



(single servers at the bottom)

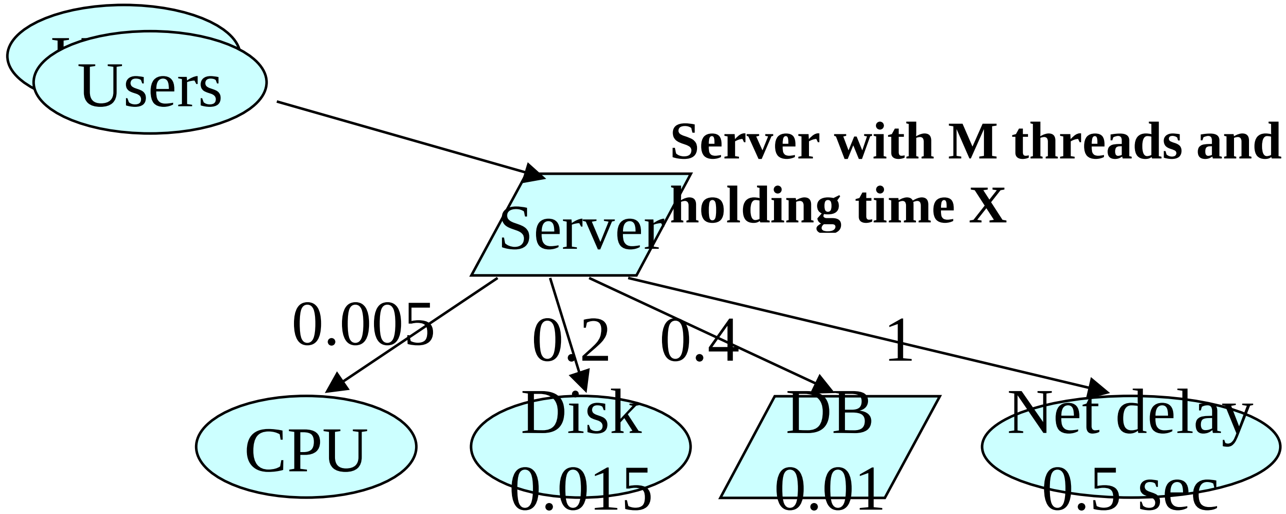
(1 sec demand at each server, one request to each lower task)



Carleton UNIVERSITY

Bottleneck: Results for a web server with net delay

N Users with a thinking time of 5 sec.

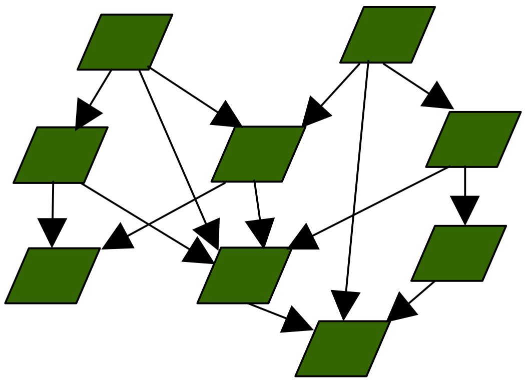


Server with M threads and holding time X

N users	500	500	500	500	2000	2000	2000	2000
M threads	10	30	100	inf	10	30	100	inf
X server	.512	.52	.52	.52	.512	.515	.55	4.99
f thruput	19.5	58.2	90.6	90.6	19.5	56.7	180	200
W user wait	20.6	3.6	0.51	0.5	97.6	29.4	6.1	5
U server	10	30	47	47	10	30	100	1000
U net	9.7	29.1	45.3	45.3	9.7	29.1	90.2	100
U CPU	.097	.29	.45	.45	.097	.29	.90	1.0

Multiple independent bottlenecks

- there may be a web of servers and interactions



- perhaps there are multiple bottlenecks?
  - in a flat queueing network there can be as many independent bottlenecks as there are chains of customers
  - each is an independent limitation on chain throughputs
- in a layered queueing network there are only a few independent throughputs... e.g. the top-layer tasks

Carleton UNIVERSITY

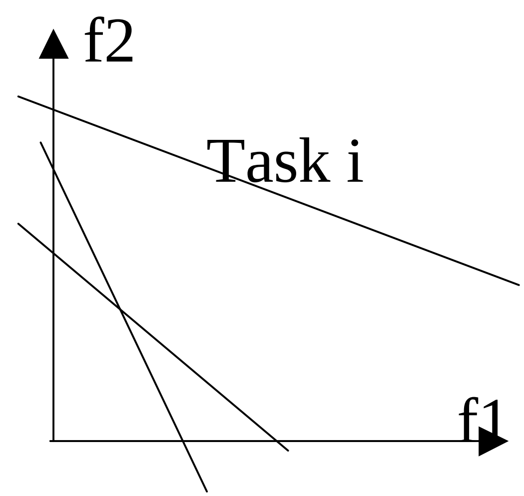
Using bound relationships in layered queues

- reference throughputs f_r at the "user" top-layer tasks
- all other throughputs (fe at entry e) are proportional, $f_e=\Sigma_r\, a_{re}\, f_r$
- no-waiting service time of entry e is x_e which can be computed recursively using nested delays
- reference throughputs must satisfy the utilization constraint $U_i < M_i$,

$$\Sigma_{e \text{ in Task } i}\ f_e\, x_e < M_i$$

$$\Sigma_r\ f_r\, \Sigma_{e \text{ in Task } i}\ a_{re}\, x_e < M_i$$

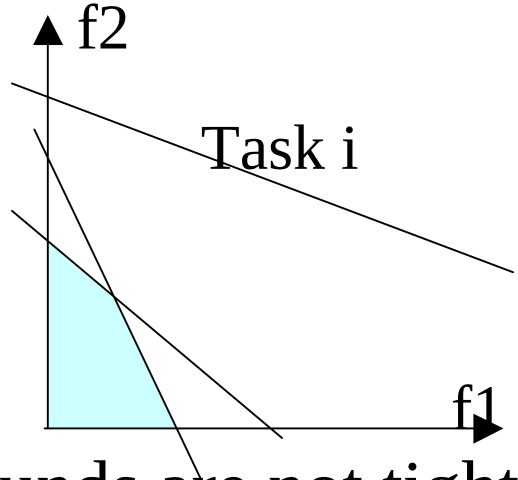
$$\Sigma_r\ f_r\ K_{ir} < 1$$



- *"rendezvous nets" paper 1995*

Bound relationships are only a crude guide to bottleneck location

- resources giving bounds that touch the feasible region are likely candidates
  - other bounds, e.g. for Taski, are prevented from saturating



- however the bounds are not tight
  - because they ignore queueing delays at intermediate levels
  - since queueing delay can create a bottleneck... it really needs a full queueing solution

Converting results to software implications

- long response or service times can be reduced by
  - parallelizing some operations
  - balancing and reduced variability in parallel paths
  - latency masking (e.g. by pre-fetching)
  - optimistic design
  - removing bottlenecks within the response
- bottlenecks can be reduced by...
  - host demands reduced, server demands reduced
  - demands made more deterministic
  - changed allocations
  - replication, threading
  - task splitting for concurrent access (servers, pipelining...)
- navigation of sensitive points (drill-down) ***(Maps and Paths paper 1995)***

Carleton UNIVERSITY

Acting on the recommendations

- changes to software architecture and detailed design
- ***reducing demands*** is a well studied topic, e.g. Smith and Williams books
  - detailed code changes based on hot spots, locality, early binding of references
  - caching
- some of the other recommendations relate to ***"performance antipatterns"*** described by the same authors (WOSP 2000)
  - the "one lane bridge" is any bottleneck task
  - the "god class" is a task that can be split into smaller parts

Carleton UNIVERSITY

Summary

- the layered queueing model is a middle ground between software structure and queueing networks
  - default stochastic semantics have few parameters
  - scalable extended queueing canonical form
- fairly direct traceability of
  - software tasks to performance model objects
  - object interactions into model interactions
  - demands
  - results connected back to software observations
- similarity between model results and measurements on the software

Carleton UNIVERSITY

Where is this area going?

- solvers still pose open questions
  - improvements to accuracy and to features
- support for building models
  - models from UML
  - models created or updated from monitoring
- integration with discrete-state modeling methods
  - failure states (IPDS paper 98, others)
  - adaptation and variable configurations
  - submodels for inter-task protocols, using Petri Nets etc
  - submodels for more accurate delay distributions
- optimization (e.g. Sigmetrics 2001)