Reimplementation of the ATLAS Online Event Monitoring Subsystem

Ingo Scholtes
Summer Student
University of Trier

Supervisor: Serguei Kolos

TDAQ ATLAS

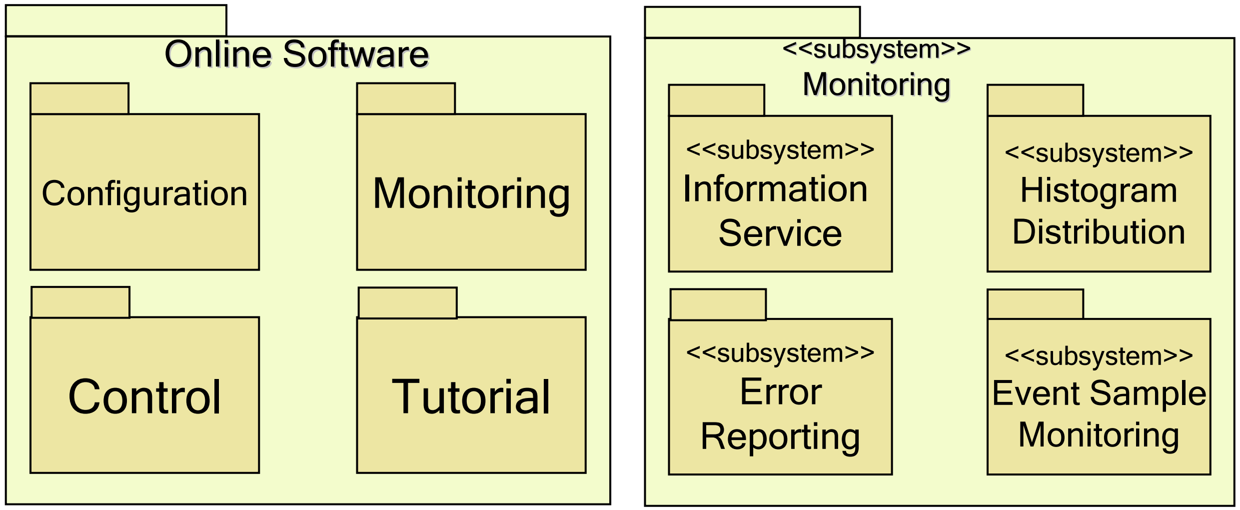
Outline

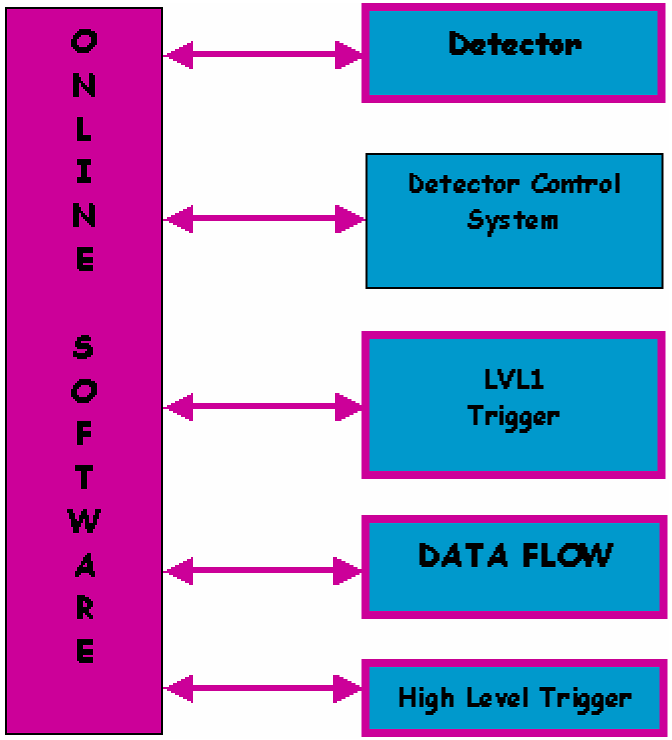
- Online Monitoring
- Current implementation and its drawbacks
- My reimplementation
- Performance comparison
- Conclusion

TDAQ

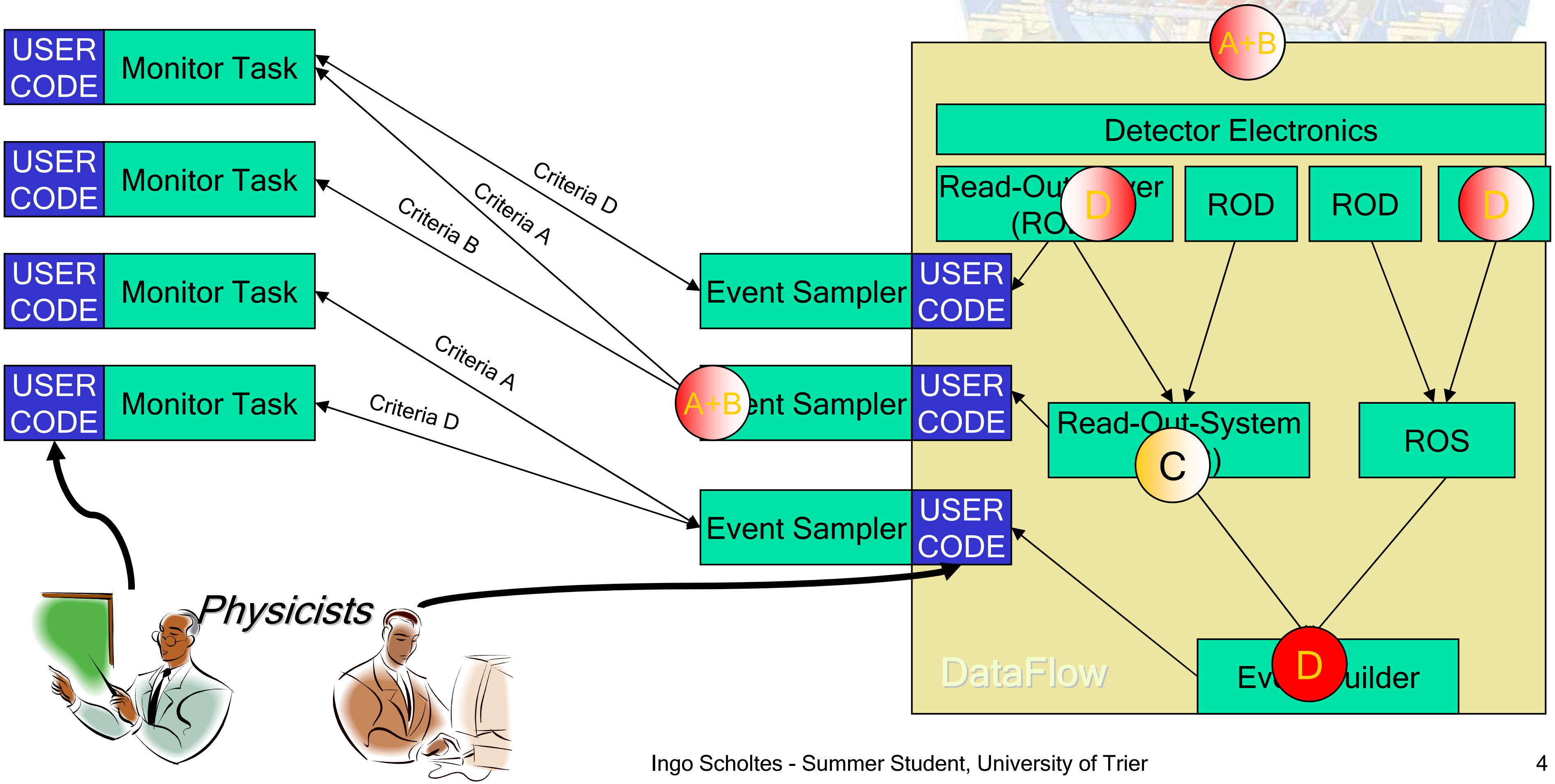
Getting the context: Atlas Online Software

- System of the Atlas Trigger DAQ Project
- Main purpose: configure, control and monitor data acquisition system
- Provides a GUI, which allows to control the data acquisition system
- "Glue" of several TDAQ sub-systems
- Open Source project

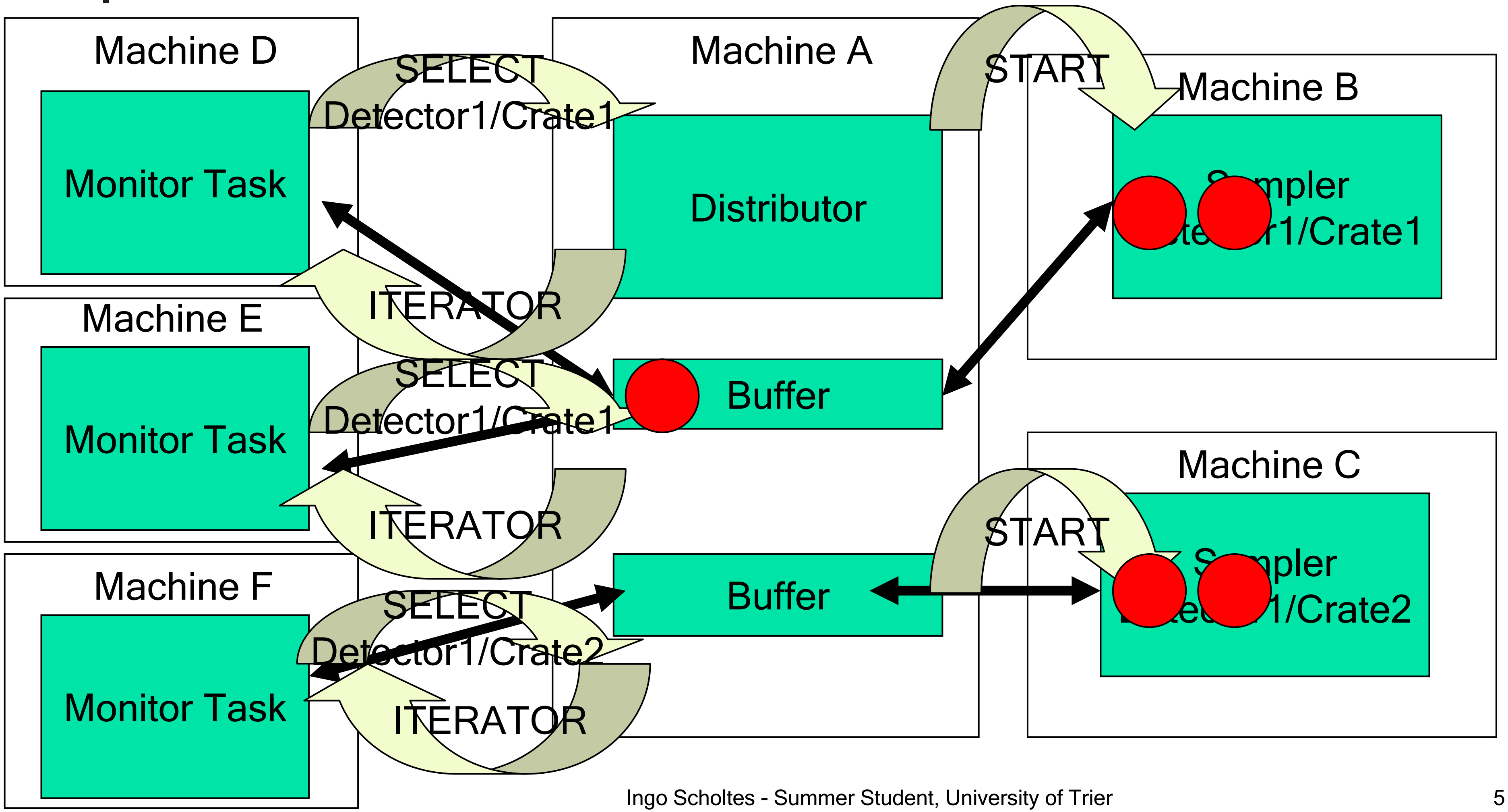




A brief overview of the ATLAS Online Event Monitoring Subsystem



The current implementation in detail



Drawbacks…

- Scalability problem due to bottleneck in machine A
- Monitors will not notice if sampler crashed
  - They just stop receiving events…
- Users will have to worry about thread management in sampler
  - Start thread on StartSampling
  - Cleanly exit it on StopSampling
  - Often causes problems

…and what we learn from them

- Core of scalability problem: central distributor
- Bottleneck due to…
  - routing of events through central distributor
  - multiple distribution of identical events to different monitors

Implementation requirements

- Platform independent C++
- Using Online Monitoring IPC based on CORBA (omniORB 4)
- minimal and deterministic effect on the data flow system performance
- High scalability
- Get rid of all drawbacks… ;-)

What is really crucial?

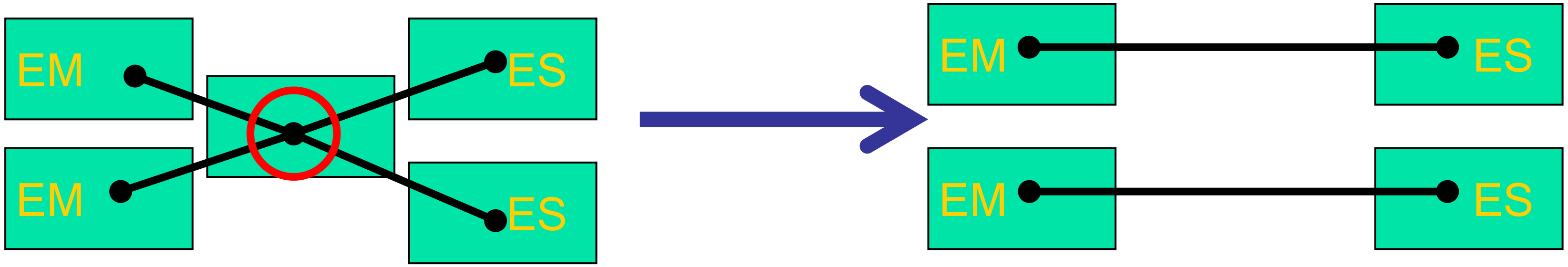
- Sampler has to decide about criteria
  - → saves a lot of bandwidth
- Sampler has to send each event once (per selection criteria)
- Distributor necessary to protect sampler from inrushing monitors (gatekeeper function)

Basic improvement ideas

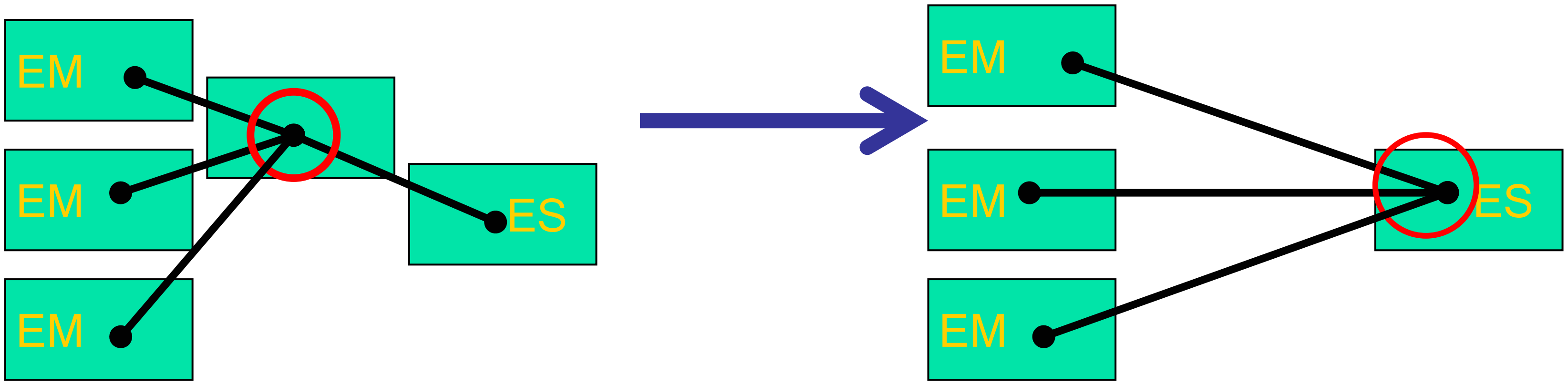
- Get rid of distributor for communication → P2P
- Moving load to monitors for means of scalability
  - Current: share bandwidth, accumulate load
  - Idea: share load, accumulate bandwidth ;-)
  - Distributor only for connection management and error recovery
- Keeping only crucial things in sampler
  - Criteria decisions
  - One-time sending of each event (→ at least one connection per sampler/criteria)
- Sampler thread management
  - Start sampling thread with first subscription
  - End sampling thread with loss of last subscription
  - User code not aware of threads

Getting rid of the distributor bottleneck

- can be obtained by using P2P paradigm
  - One monitor per sampler

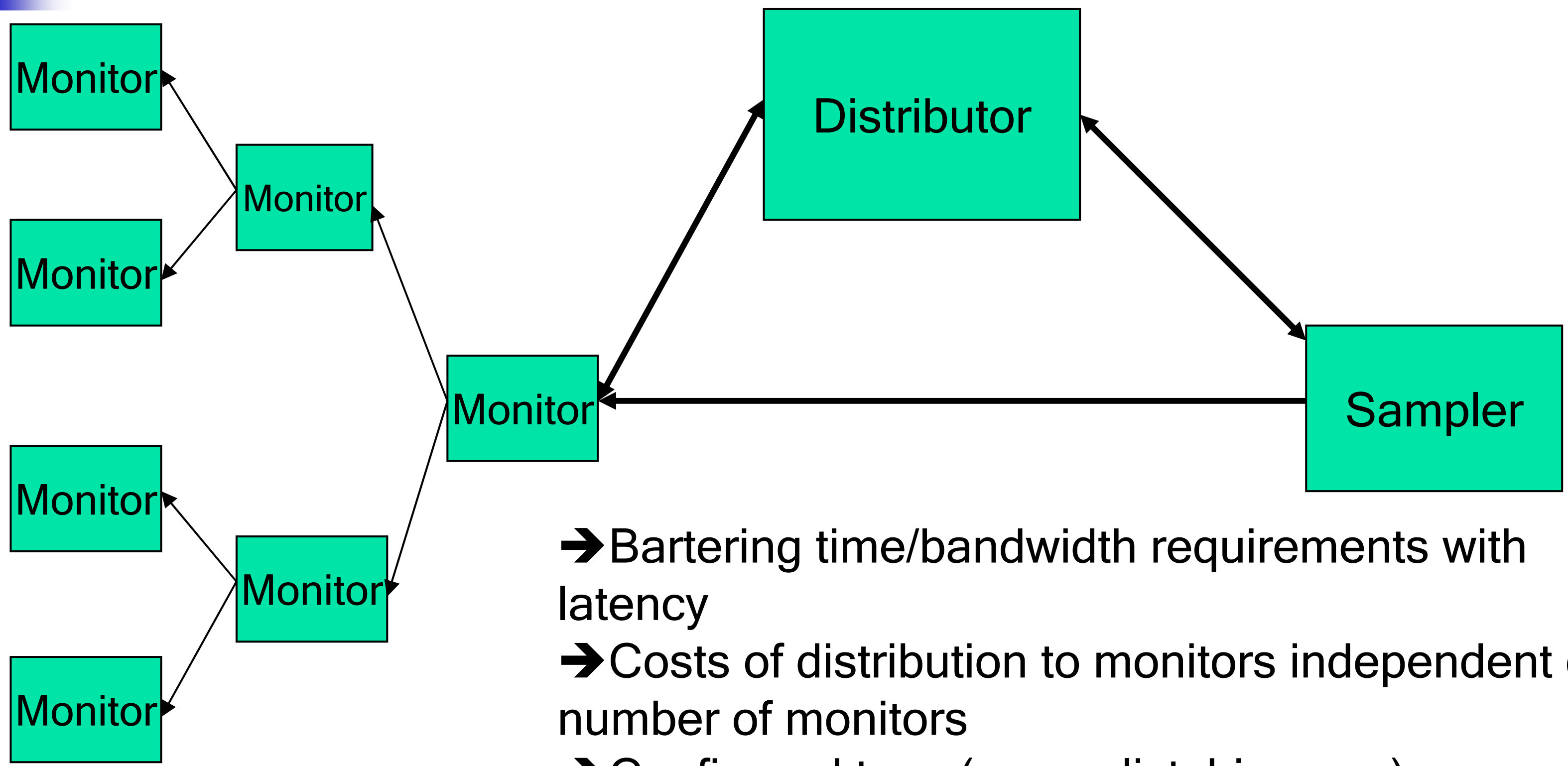


  - Multiple monitors per sampler?



We need a way to prevent bottleneck here!

Introducing the monitor tree



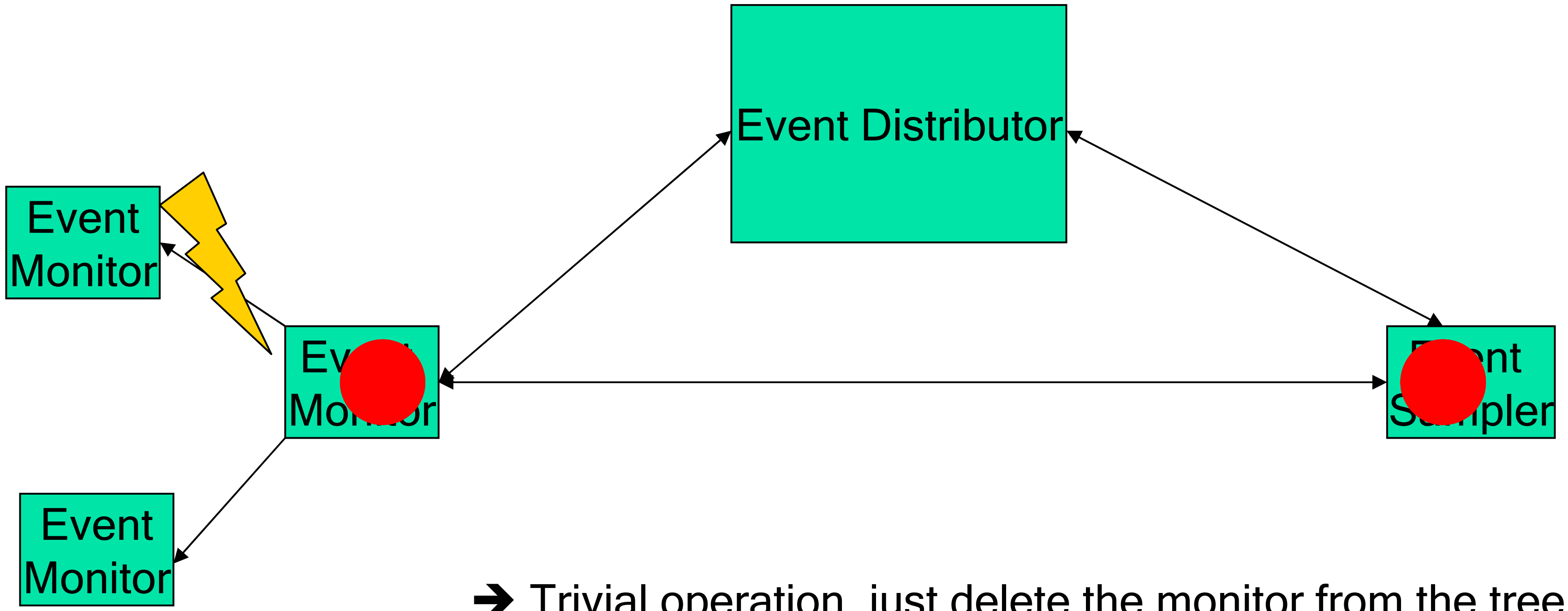
Example: Binary Monitor tree

➔Bartering time/bandwidth requirements with latency
➔Costs of distribution to monitors independent of number of monitors
➔Configured type (unary=list, binary, …) influences latency/bandwidth tradeoff
➔But: new problems arise with this structure

Problem 1: exit of monitors

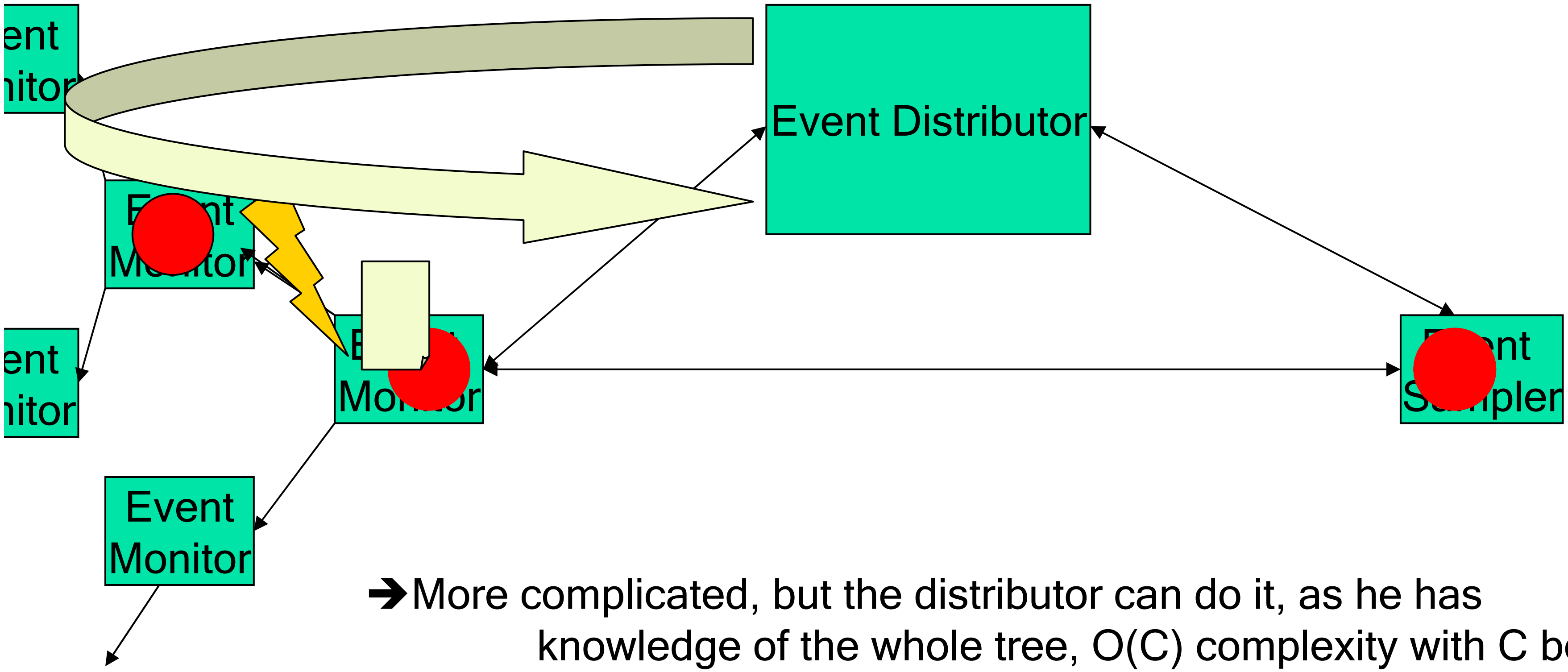
- Each monitor acts as a sampler for his children
- Exits/crashes of monitors critical…
- We have to distinguish between different types of exits
  - Leaf monitor → trivial
  - Monitor with outdegree > 0 → more complicated
  - Root monitor → critical

Solutions - Leaf monitor exit



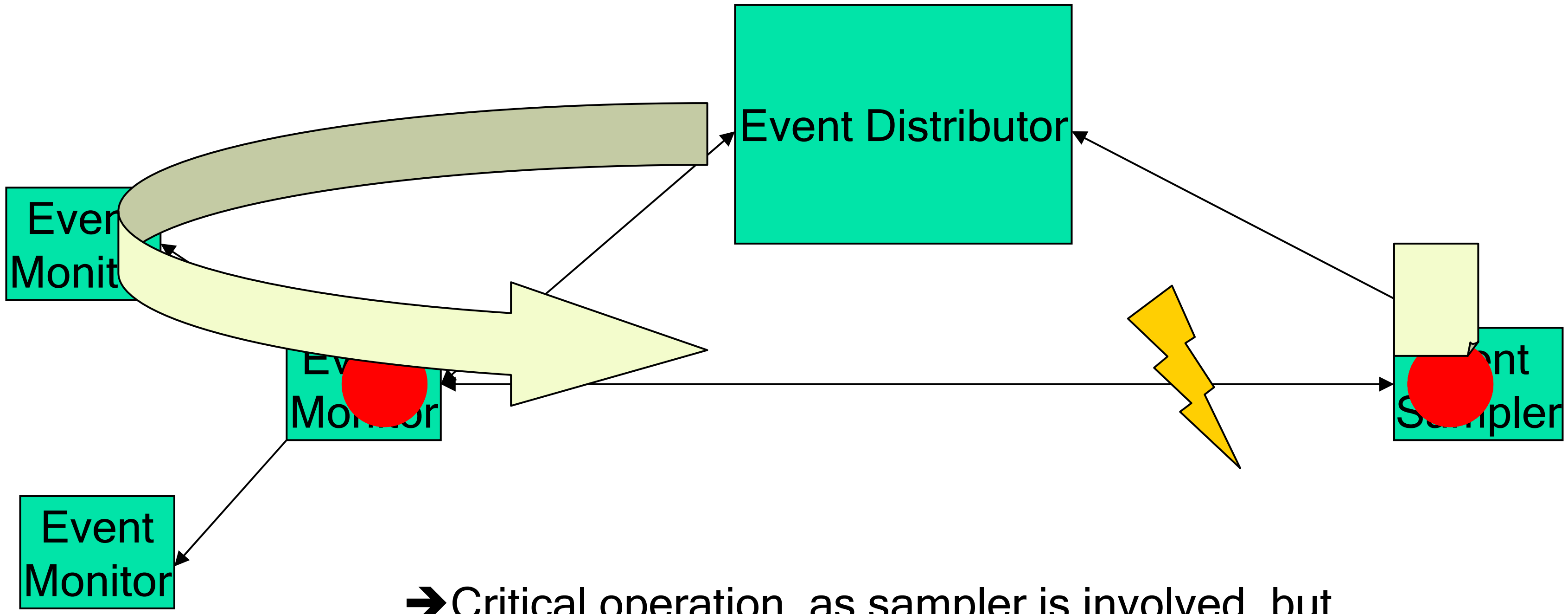
➔ Trivial operation, just delete the monitor from the tree!

Solutions - Monitor with outdegree > 0 exits



➔More complicated, but the distributor can do it, as he has knowledge of the whole tree, O(C) complexity with C being constant maximum number of children

Solutions - Root monitor exit



➔Critical operation, as sampler is involved, but possible to do it transparently for other monitors, again O(1) complexity

Problem 2: error recovery

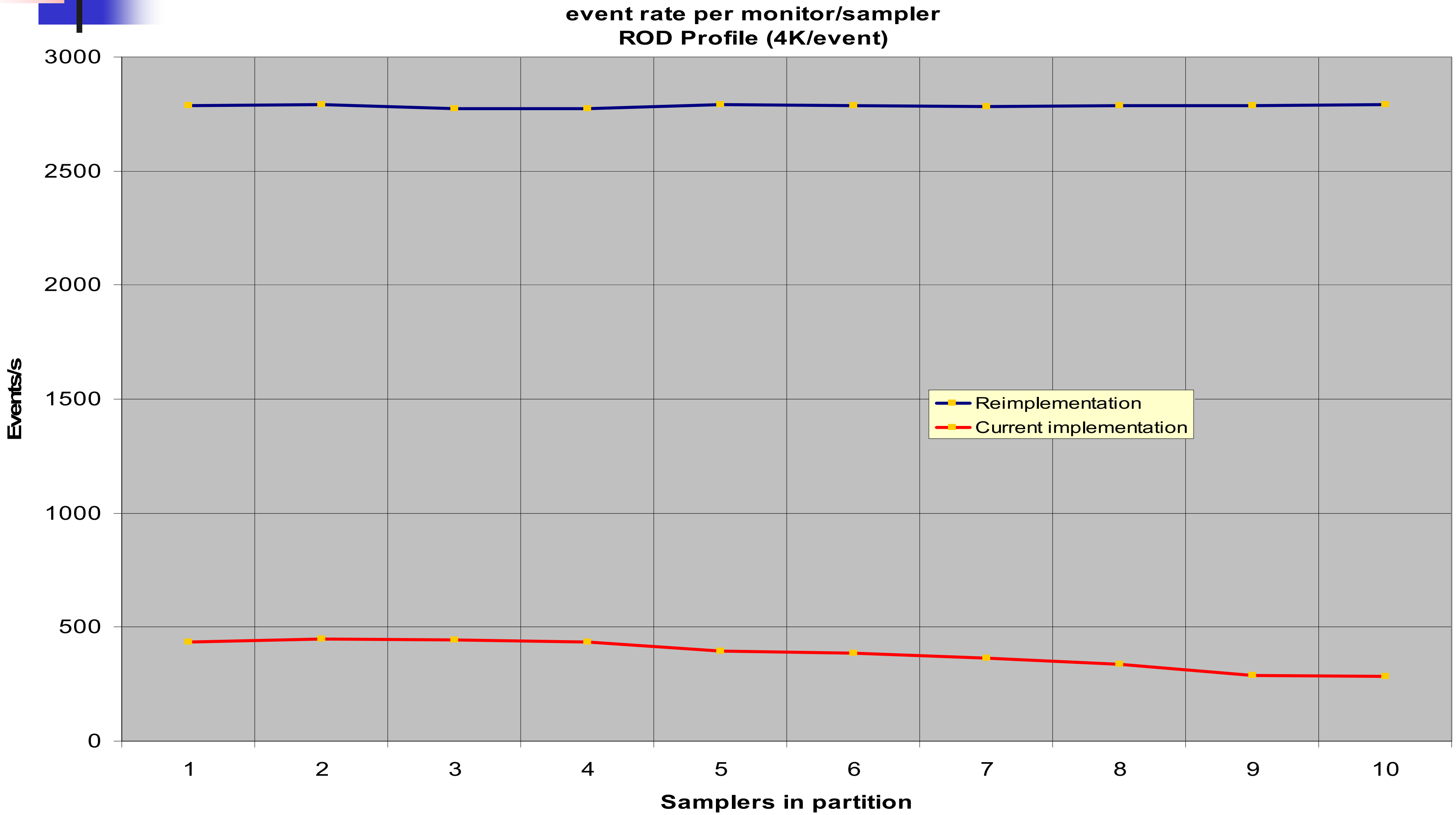
- Crash of sampler
  - Distributor pings all samplers in reasonable intervals → can notify monitors about crash
- Crash of arbitrary monitors
  - Detected like normal exit! ➔ no problem
- Crash of distributor
  - No influence on ongoing data exchange
  - Just restart…

Comparing performance…

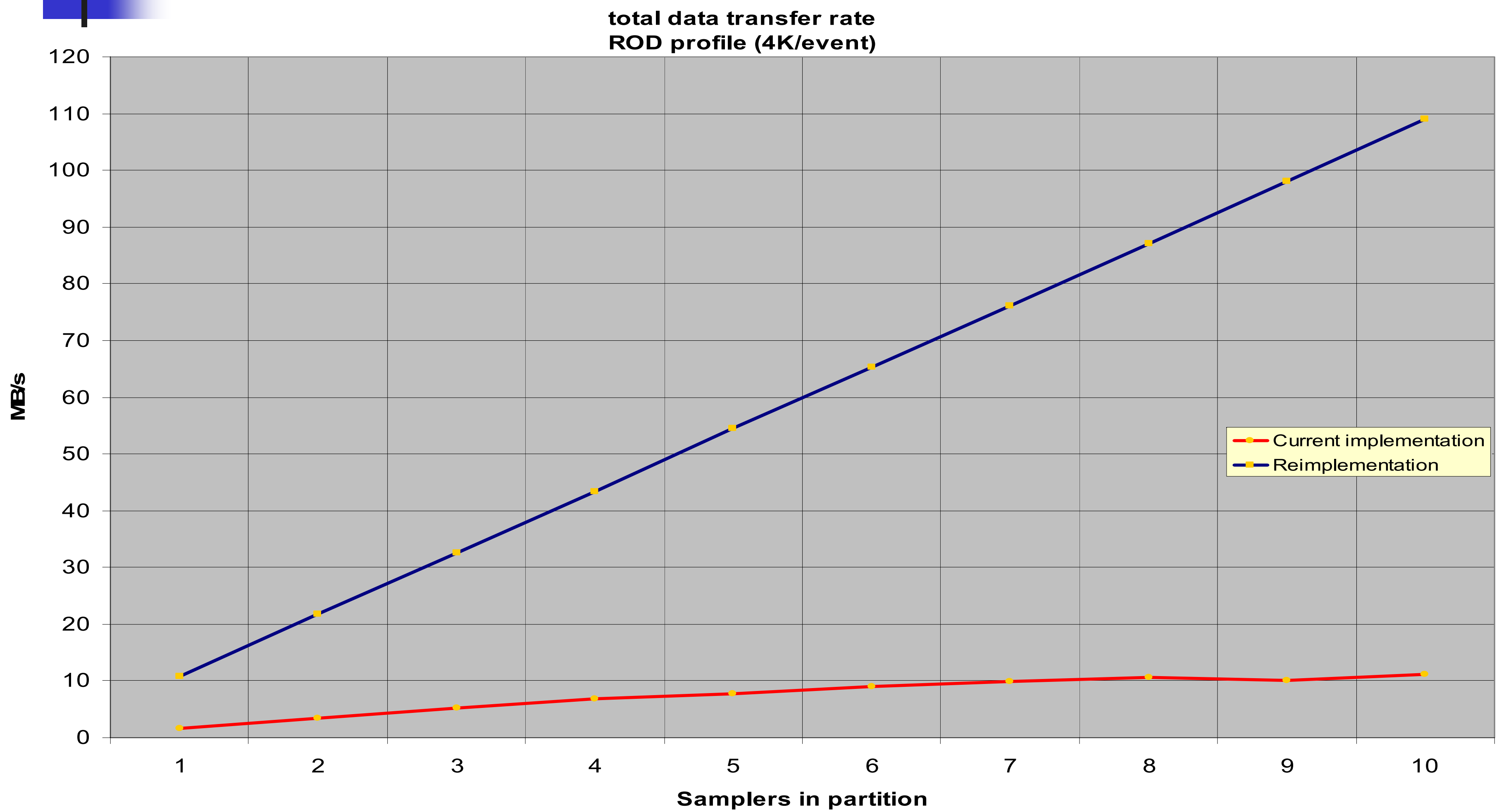
	Current implementation	Reimplementation
Sampler *i*	$2 \cdot cr_i + e \cdot cr_i = O(e \cdot cr_i)$	$2 \cdot cr_i + e \cdot cr_i = O(e \cdot cr_i)$
Monitor *i*	$2 + e = O(e)$	$2 + e + ch_i \cdot e \underset{\substack{ch_i \le C \\ \text{constant}}}{=} O(e)$
Distributor	Init & Shutdown : $2m = O(m)$ Run : $e \cdot m + e \underbrace{\sum_{i=1}^{s} cr_i}_{\le m} \le 2m \cdot e = O(m \cdot e)$	Init & Shutdown : $2m = O(m)$ Run : 0

cr_i = # criteria in sampler i
s = # samplers
ch_i = # children of monitor i
C = max. children/monitor
e = # sampled bytes
m = # monitors

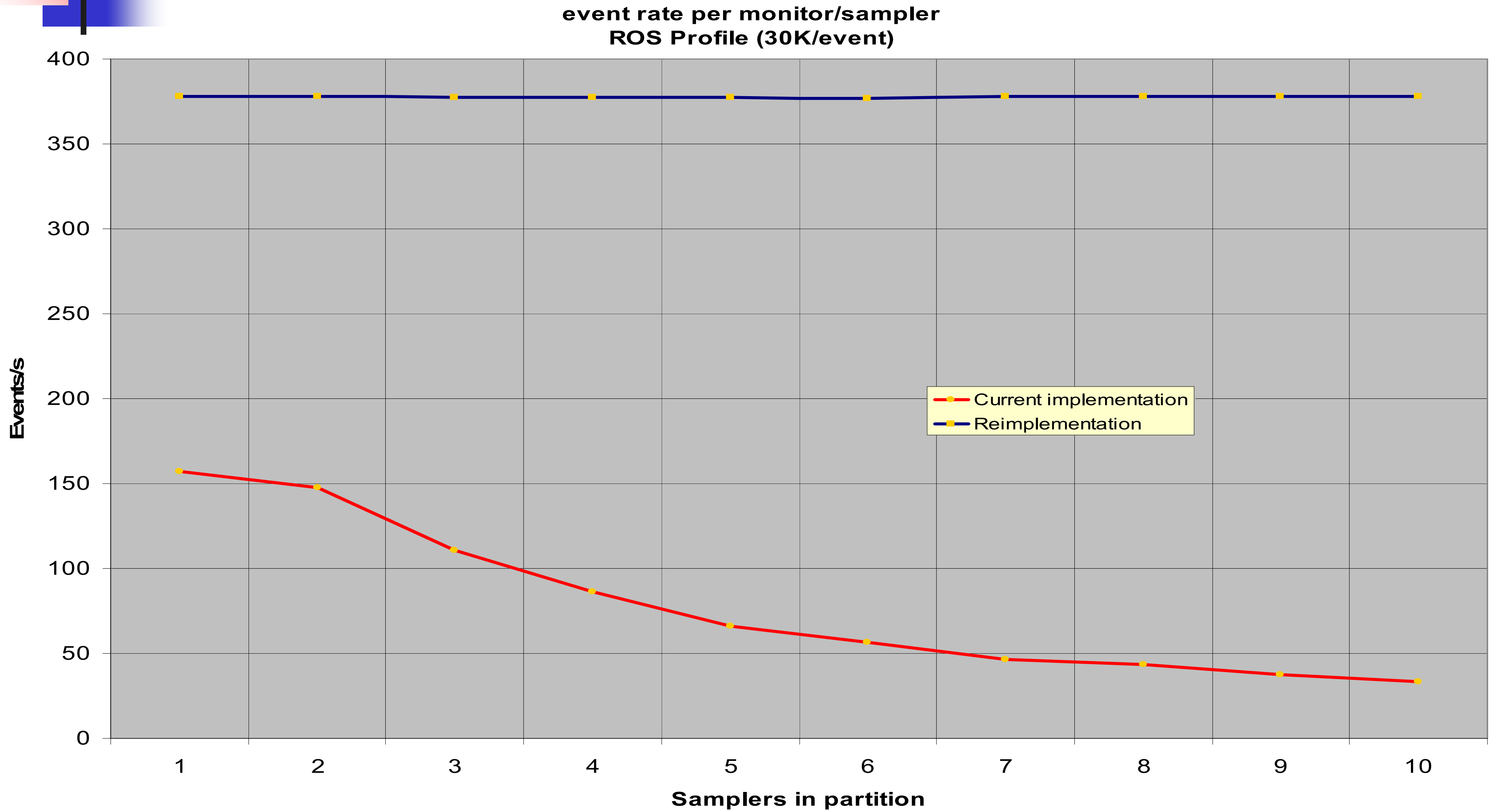
ROD Profile (10.000 Events @ 4K)



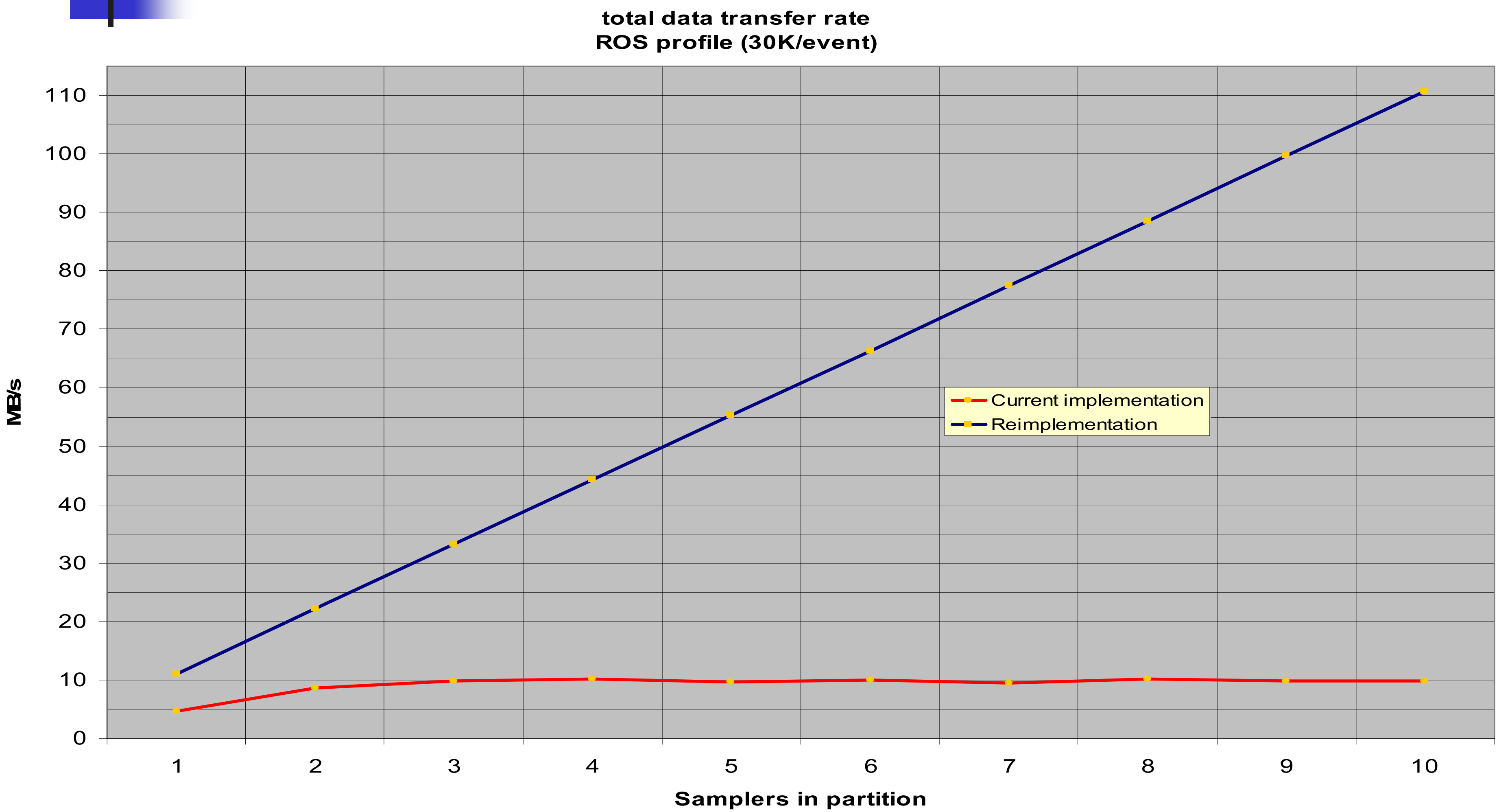
ROD Profile (10.000 Events @ 4K)



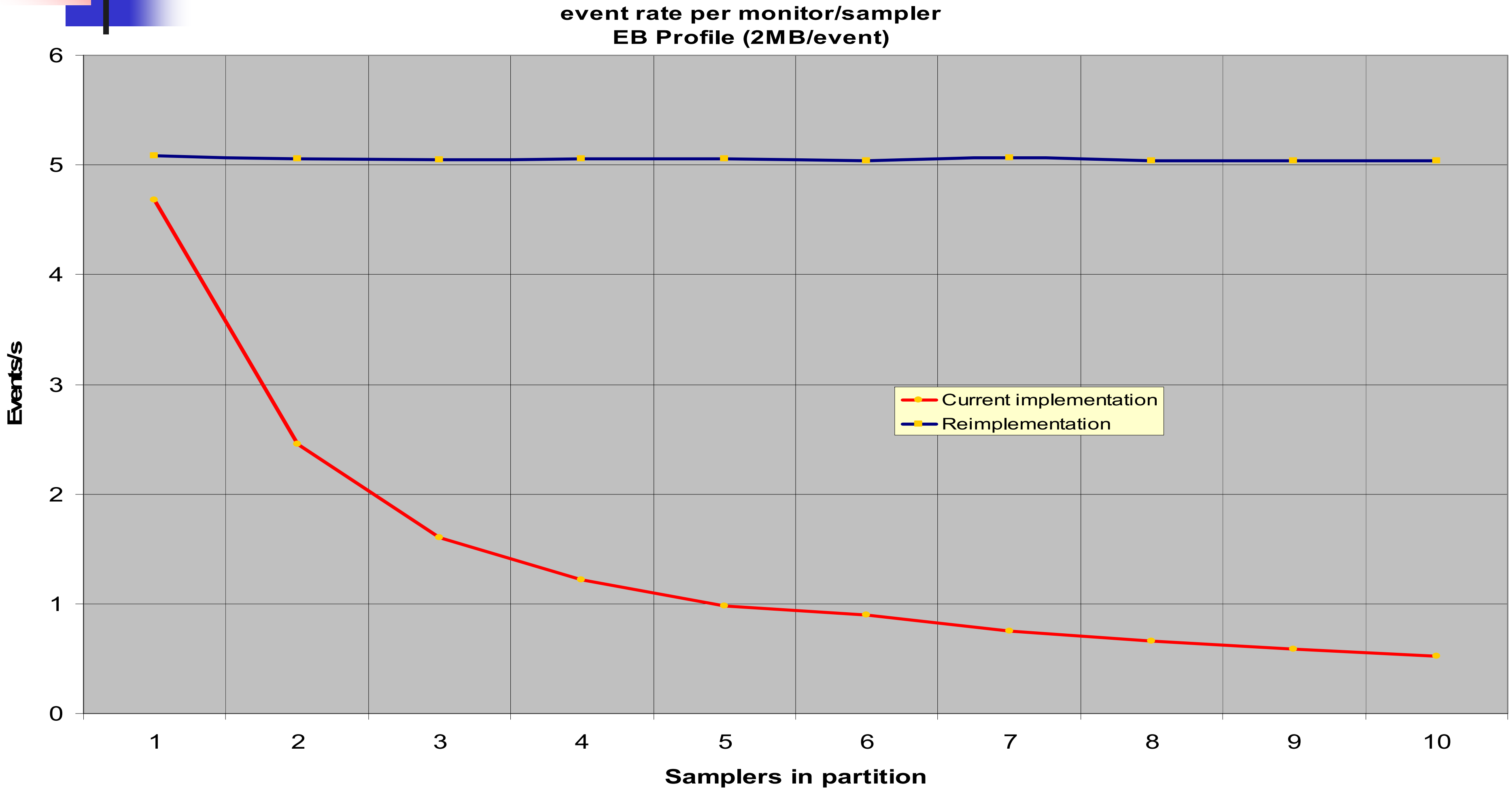
ROS profile (10.000 Events @ 30 K)



ROS profile (10.000 Events @ 30 K)

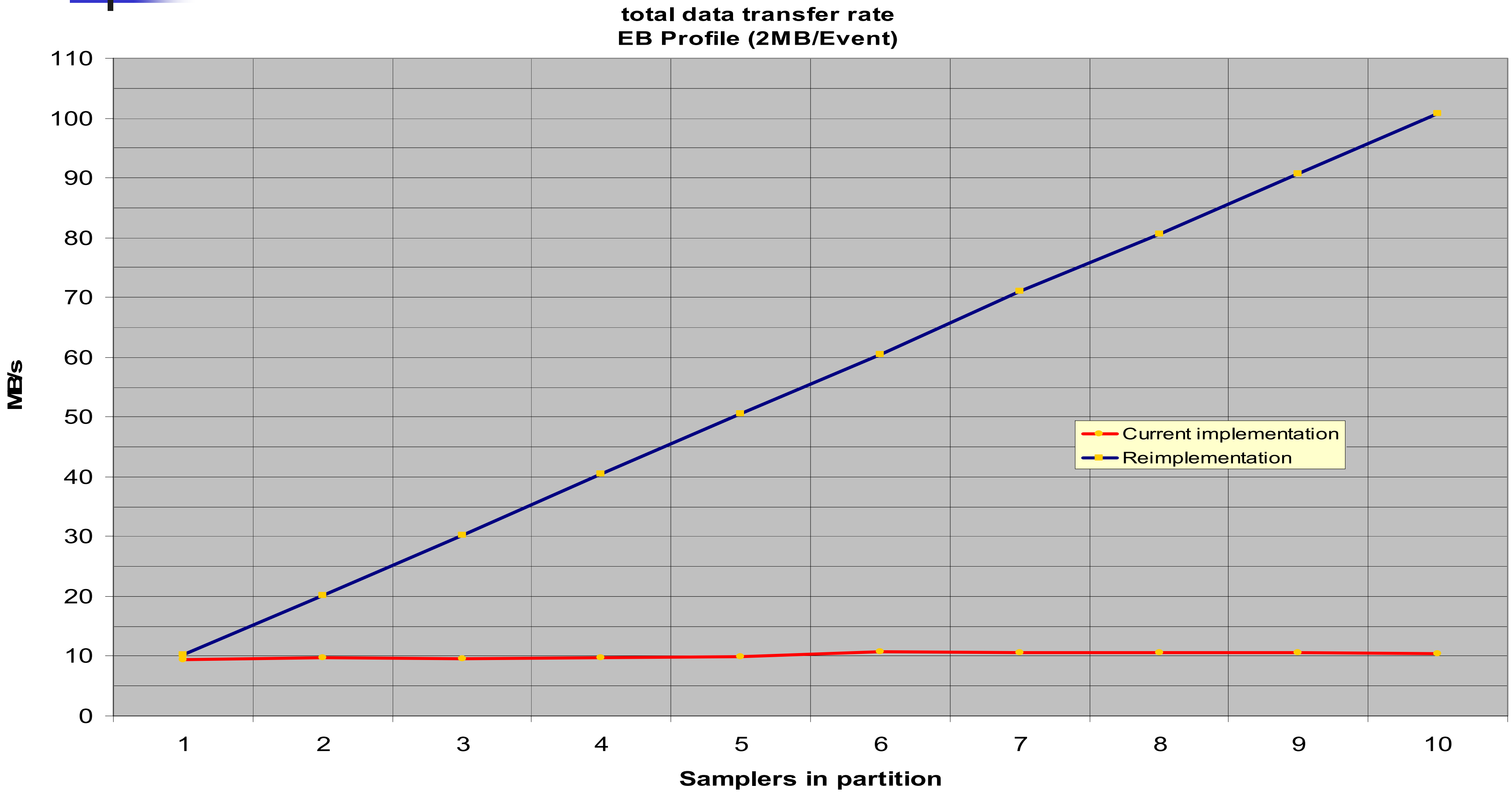


EB profile (100 Events @ 2MB)



TDAQ

EB profile (100 Events @ 2MB)



Conclusion

- Reimplementation fulfills all needs
  - Improved speed
  - As seen: Optimal scalability (constant!)
  - Enhanced error recovery
  - Configurable tradeoff between latency and CPU/bandwidth requirements (tree type unary, binary, …)
  - Users do not need to care about thread management

Thanks…

- …for your attention!
- Questions?
- Criticism?