How can we support Grid Transactions?

Towards Peer-to-Peer Transaction Processing

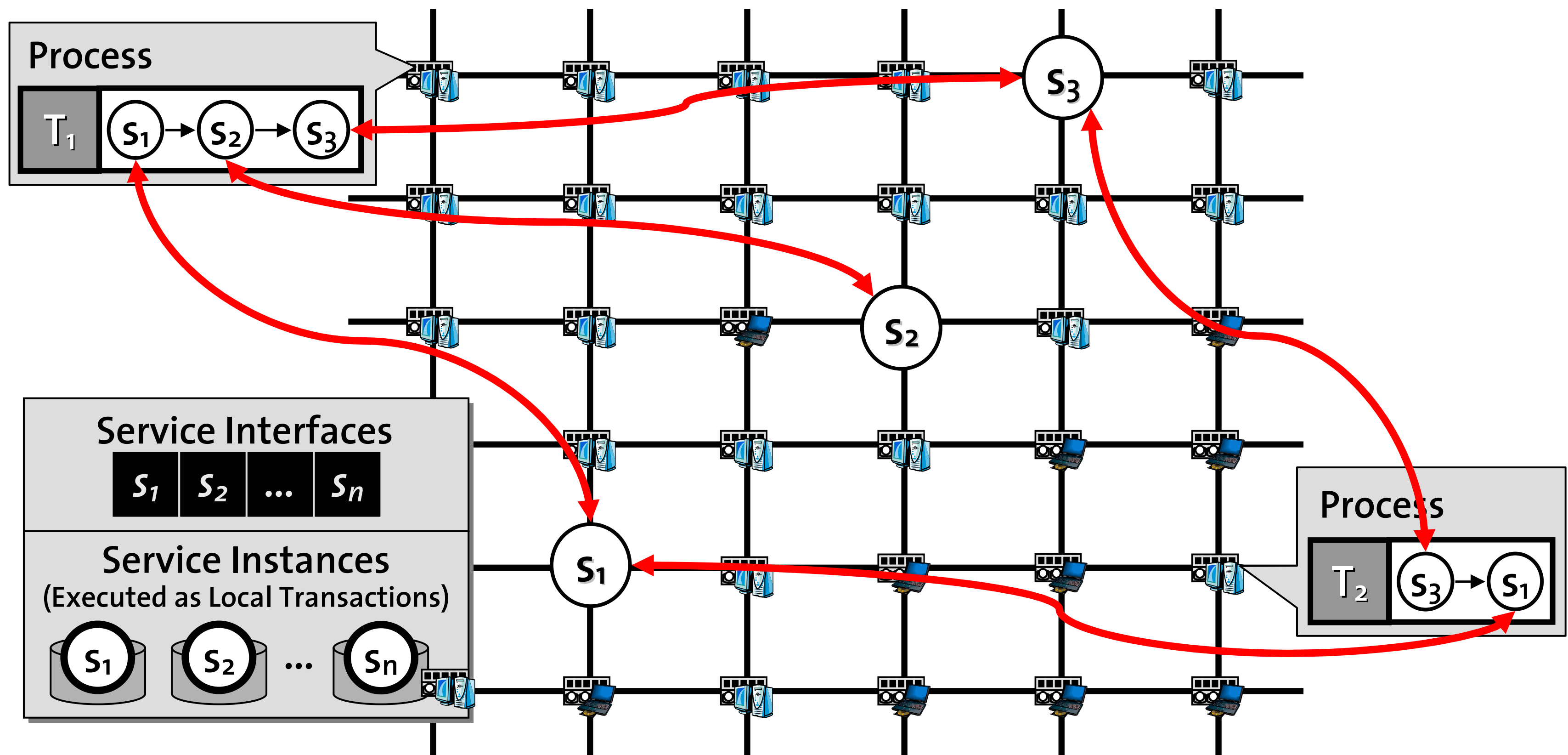
Can Türker, Klaus Haller, Christoph Schuler, Hans-Jörg Schek

ETH Zurich
Institute of Information Systems
Database Research Group

tuerker@inf.ethz.ch

Motivation

- Grid resources (peers) provide services
- Processes composed of service invocations
- Dependencies between services → transactional guarantees needed



Concurrency Control & Recovery in the Grid

- **Composite services executed as multi-level transactions**
- **No central coordinator**
- **Semantic concurrency control & recovery**
  - Service level instead of data level
  - Conflicts defined regarding service semantics
- **Long-running transactions (workflows/processes)**
  - Non-blocking
  - Partial rollback

Distributed Concurrency Control

- **Locking Approaches**
  – 2PL/2PC combined with distributed deadlock detection (or timeout)
  → Problem: blocking protocol

- **Certifier Approaches**
  – Failure detection postponed until commit time
  → Problem: many rollbacks (expensive in case of long-running transactions)

- **Timestamp Ordering Approaches**
  – Entrance to system determines correct execution order on peers
  → Problem: many unnecessary rollbacks

- **Serialization Graph Approaches**
  → Problem: cycle detection & cascading rollbacks
  → But costs of cycle detection not significant w.r.t. long-running transactions

Our Approach

Observation:

- A transaction may only commit if all transactions on which it depends have committed

Approach: Decentralize serialization graph testing

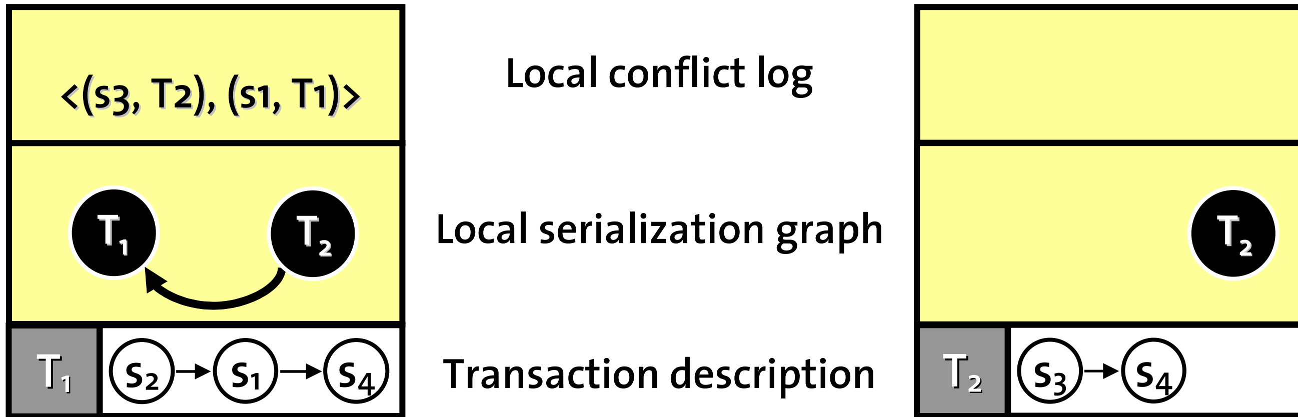
- Equip transactions with necessary dependency knowledge such they can decide to commit without a global coordinator
- Transactions require knowledge about
  - directly preordered transactions

    → *from peers (to ensure correctness)*
  - transitively dependent transactions

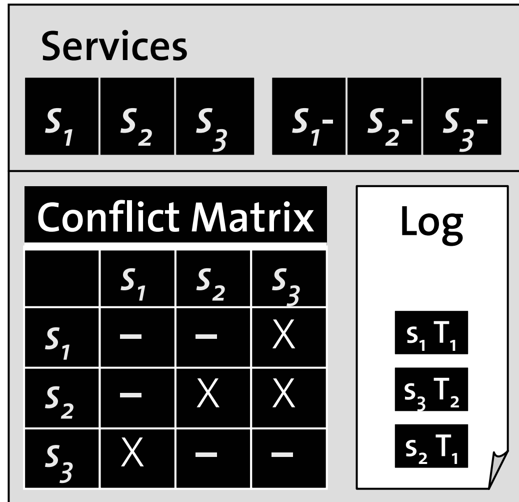
    → *from transactions (to detect cyclic dependencies)*
- Local, incomplete, not necessarily up-to-date knowledge

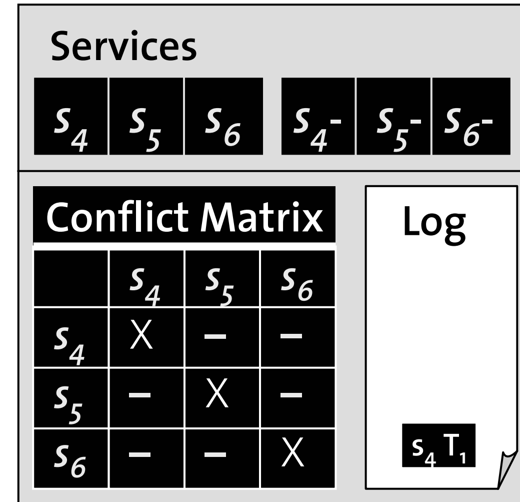
System Model

Transactions



Peers

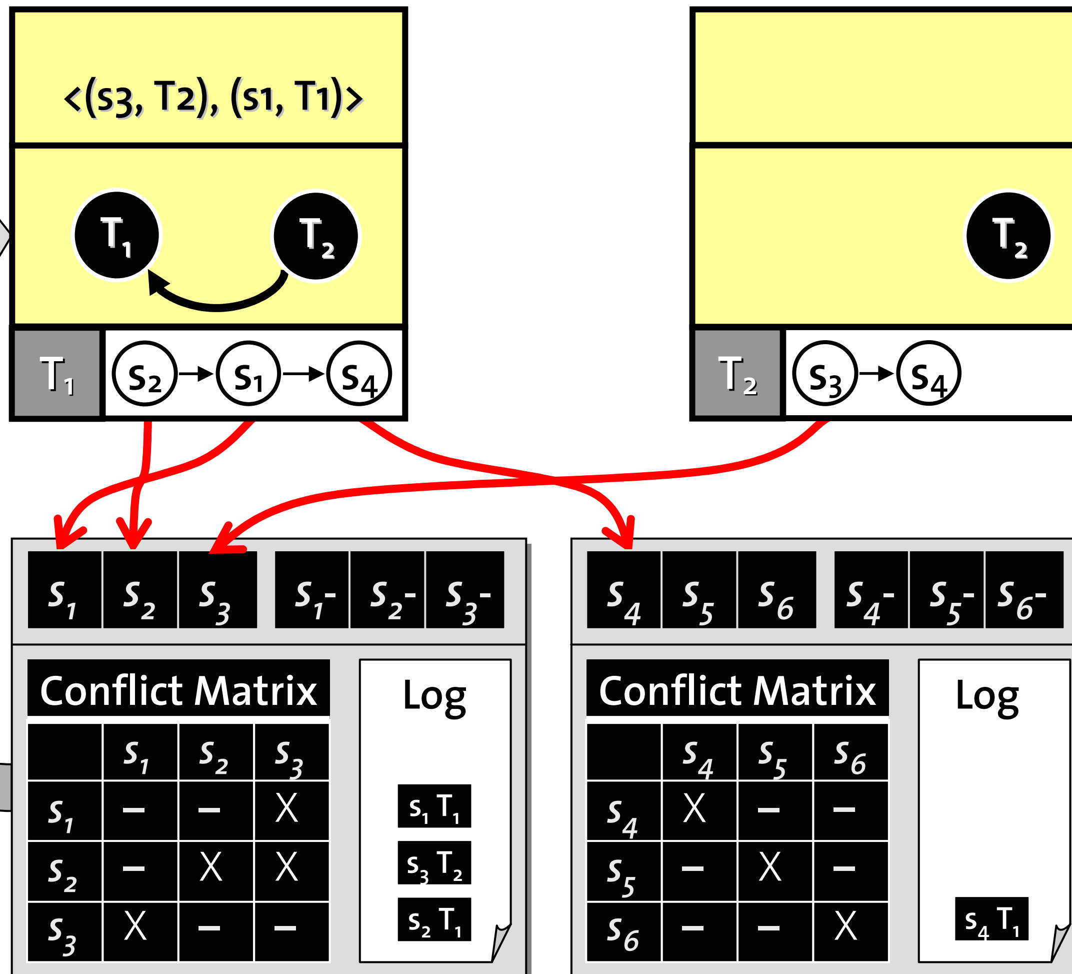




Preventing Incorrect Schedules

Rule: Transaction must not commit before all preordered transactions have committed

⇒ Transaction receives relevant conflicts as part of service invocation reply



Peer detects & informs about conflict

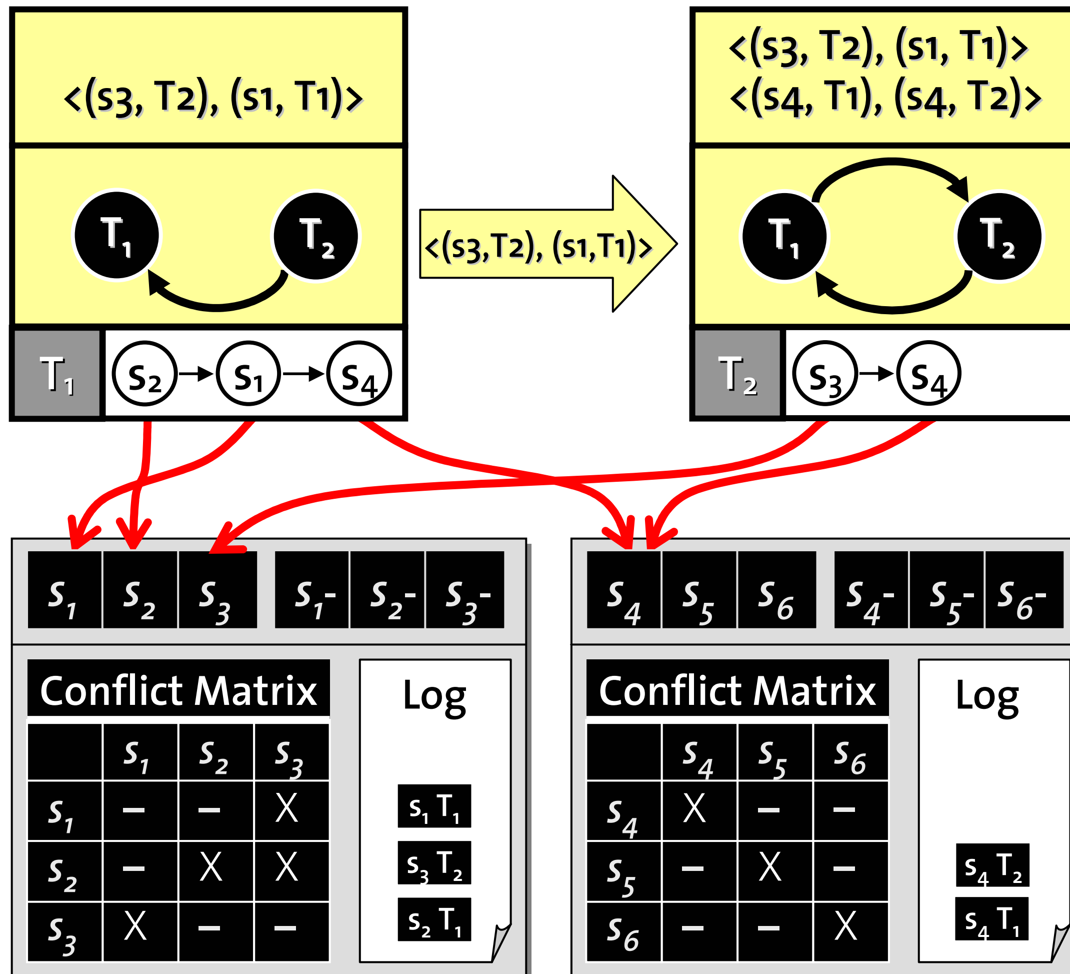
T_1 must wait for the commit of T_2!

Non-serializable schedules cannot occur!

Detecting Cyclic Waiting Situations

Observation: Cyclic waiting situations cannot be detected with local knowledge only

⇒ Push paths to preordered transactions

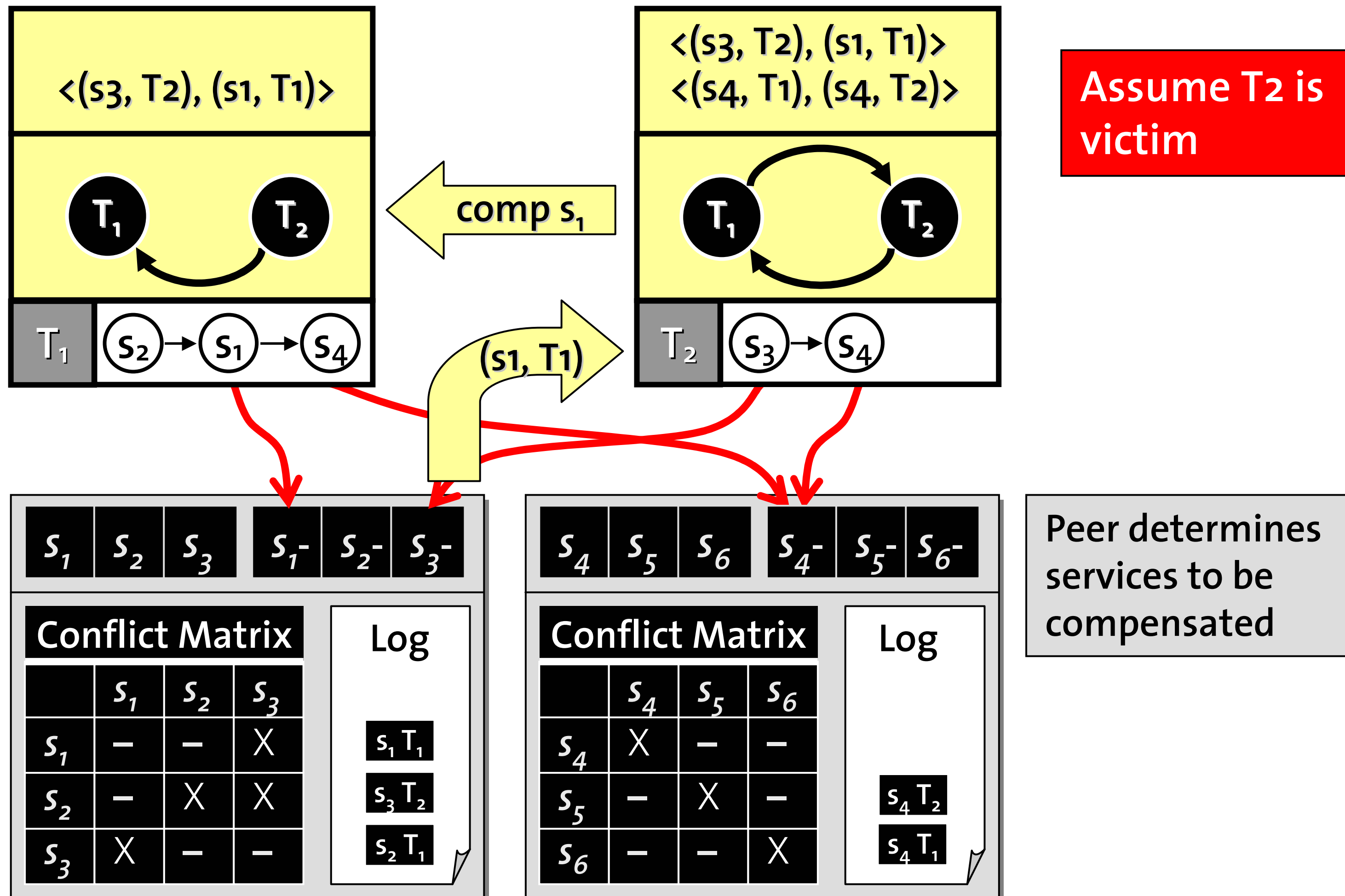


Cycle detected!

Solving Cyclic Waiting Situations

Rule: If cycle detected, rollback partially until cycle disappears and then restart

⇒ Peer determines conflicting service invocations to be compensated



Experiments: DSGT vs. S2PL

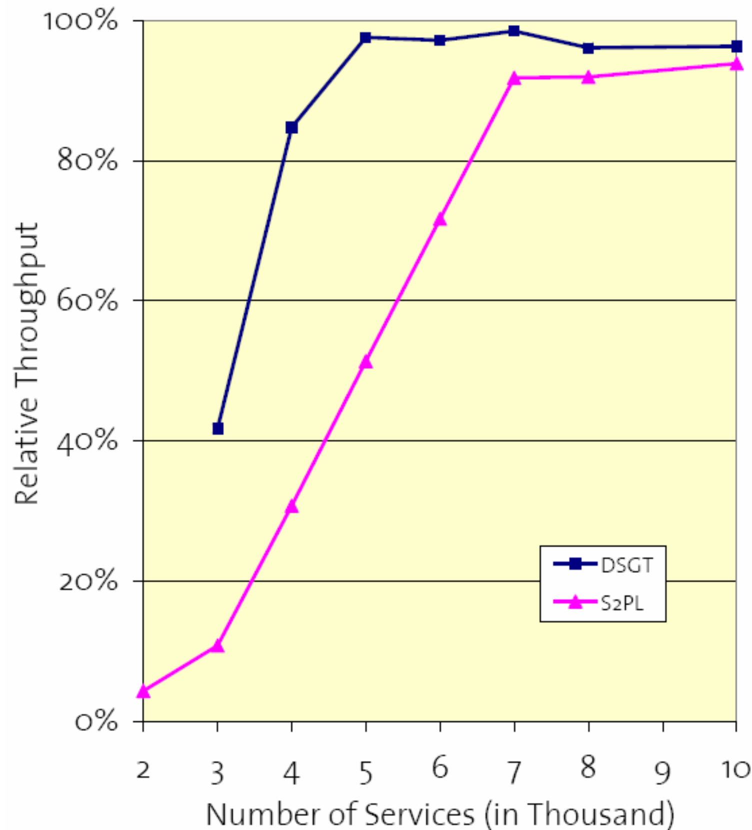
Based on IBM WebSphere

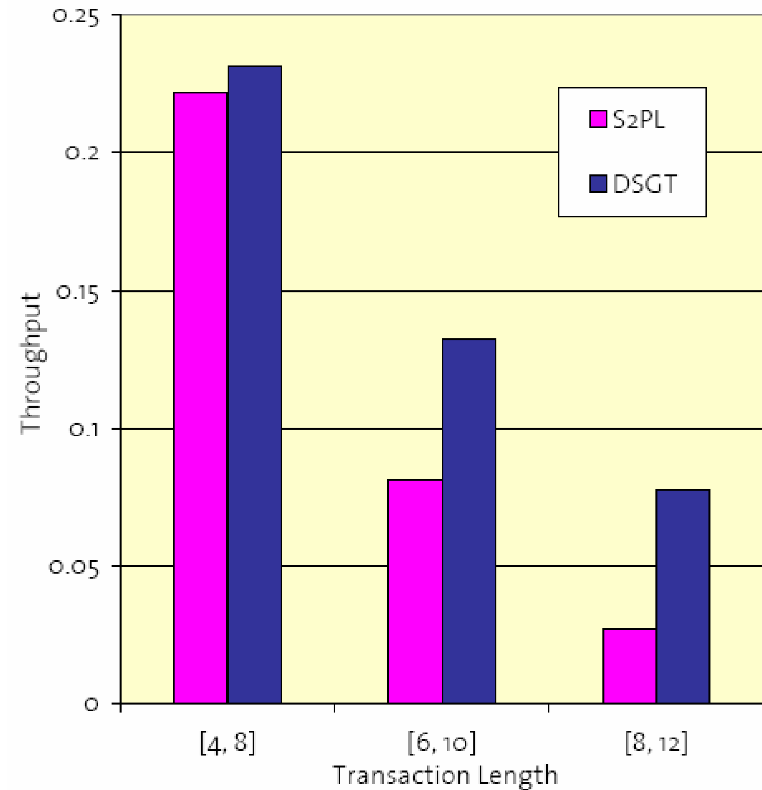
Five hosts each always running 20 active transactions

Transactions consists of 8-12 service invocations

Service durations 2 seconds

Restart delay 0-20 seconds





Conclusions and Outlook

- **Decentralized Concurrency Control & Recovery**
  - Based on "optimistic" serialization graph testing
  - For service-oriented, peer-to-peer systems
- **Results**
  - Global correctness relying only on local, incomplete knowledge
  - Partial rollback reduces costs of cascading aborts
  - DSGT useful for long-running transactions (outperforms 2PL)
- **Outlook**
  - Self-adapting protocols
  - Grid partitioning