182.703: Problems in Distributed Computing  
(Part 3)  
WS 2023  

Ulrich Schmid  
Institute of Computer Engineering, TU Vienna  
Embedded Computing Systems Group E191-02  
s@ecs.tuwien.ac.at

TU Wien Informatics
Content (Part 3)

- The Role of Synchrony Conditions
  - Failure Detectors
  - Real-Time Clocks

- Partially Synchronous Models
  - Models supporting lock-step round simulations
  - Weaker partially synchronous models

- Dynamic Networks
The Role of Synchrony Conditions
Recall Distributed Agreement (Consensus)
Recall Consensus Impossibility (FLP)

Fischer, Lynch und Paterson [FLP85]:

“There is no deterministic algorithm for solving consensus in an asynchronous distributed system in the presence of a single crash failure.”

Key problem:
Distinguish slow from dead!
Dolev, Dwork and Stockmeyer investigated consensus solvability in Partially Synchronous Systems (ParSync), varying 5 "synchrony handles":

- Processors synchronous / asynchronous
- Communication synchronous / asynchronous
- Message order synchronous (system-wide consistent) / asynchronous (out-of-order)
- Send steps broadcast / unicast
- Computing steps atomic rec+send / separate rec, send
Consensus Solvability in ParSync [DDS87] (II)

- Wait-free consensus possible
- Consensus impossible
- Consensus possible for $f=1$

Communication

Global message order

Consensus Solvability in ParSync [DDS87] (II)
The Role of Synchrony Conditions

Enable failure detection • Distinguish slow from dead

Enforce event ordering • Distinguish "old" from "new"

• Ruling out existence of stale (in-transit) information
• Creating non-overlapping "phases of operation" (rounds)
Failure Detectors
Failure Detectors [CT96] (I)

• Chandra & Toueg augmented purely asynchronous systems with (unreliable) failure detectors (FDs):

  • Every processor owns a local FD module (an „oracle“ – we do not a priori care about how it is implemented!)
  • In every step [of a purely asynchronous algorithm], the FD can be queried for a hint about failures of other procs
Failure Detectors [CT96] (II)

- make mistakes – the (time-free!) FD specification restricts the allowed mistakes of a FD

- **FD hierarchy**: A stronger FD specification implies
  - less allowed mistakes
  - more difficult problems to be solved using this FD
  - But: FD implementation more demanding/difficult

- Every problem \( Pr \) has a **weakest FD** \( W \):
  - There is a purely asynchronous algorithm for solving \( Pr \) that uses \( W \)
  - Every FD that also allows to solve \( Pr \) can be transformed (via a purely asynchronous algorithm) to simulate \( W \)
Example Failure Detectors (I)

- **Perfect failure detector P**: Outputs suspect list
  - *Strong completeness*: Eventually, every process that crashes is permanently suspected by every correct process
  - *Strong accuracy*: No process is ever suspected before it crashes

- **Eventually perfect failure detector ◊P**:  
  - *Strong completeness*
  - *Eventual strong accuracy*: There is a time after which correct processes are never suspected by correct processes
Example Failure Detectors (II)

- **Eventually strong failure detector ♦S:**
  - *Strong completeness*
  - *Eventual weak accuracy*: There is a time after which some correct process is never suspected by correct processes

- **Leader oracle Ω**: Outputs a single process ID
  - There is a time after which every not yet crashed process outputs the same correct process \( p \) (the „leader“)

- Both are weakest failure detectors for consensus (with majority of correct processes)
Consensus with $\Diamond S$: Rotating Coordinator

Task $T1$:
(1) $r_i \leftarrow 0; est_i \leftarrow v_i$;
(2) while true do
(3) \hspace{1cm} $c \leftarrow (r_i \mod n) + 1; r_i \leftarrow r_i + 1; \% 1 \leq r_i < +\infty \%$

---------- Phase 1 of round $r$: from $p_c$ to all ----------
(4) if ($i = c$) then broadcast PHASE1($r_i, est_i$) endif;
(5) wait until (PHASE1($r_i, v$) has been received from $p_c \lor c \in suspected_i$);
(6) if (PHASE1($r_i, v$) received from $p_c$) then $aux_i \leftarrow v$ else $aux_i \leftarrow \bot$ endif;

---------- Phase 2 of round $r$: from all to all ----------
(7) broadcast PHASE2($r_i, aux_i$);
(8) wait until (PHASE2 ($r_i, aux$) msgs have been received from a majority of proc.);
(9) let $rec_i$ be the set of values received by $p_i$ at line 8;
\hspace{1cm} \% We have $rec_i = \{v\}$, or $rec_i = \{v, \bot\}$, or $rec_i = \{\bot\}$ where $v = est_c$ \%
(10) case $rec_i = \{v\}$ then $est_i \leftarrow v$; broadcast DECISION($est_i$); stop $T1$
(11) $rec_i = \{v, \bot\}$ then $est_i \leftarrow v$
(12) $rec_i = \{\bot\}$ then skip
(13) endcase
(14) endwhile
Why Agreement? Intersecting Quorums

Intersecting Quorums:

$n=7$
$f=3$

$p$ decides $v$    every $q$ changes its estimate to $v$
Implementability of FDs

- If we can implement a FD like $\Omega$ or $\diamond S$, we can also implement consensus (for $n > 2f$)
- In a purely asynchronous system
  - it is impossible to solve consensus (FLP result)
  - it is hence also impossible to implement $\Omega$ or $\diamond S$
- Back at key question: What needs to be added to an asynchronous system to make $\Omega$ or $\diamond S$ implementable?
  - Real-time constraints [ADFT04, …]
  - Order constraints [MMR03, …]
  - ???
Food for Thoughts

(1) Starting out from the rotating coordinator consensus algorithm with ◊S shown above, devise a consensus algorithm that uses Ω instead of ◊S. (Use Ω in the first phase to receive only from the process that is considered leader, and keep in mind that different receivers may have different leaders for some time. You thus need some additional effort to make their estimates the same if somebody decides early.)

(2) Sketch the proof that your algorithm works correctly in a system of $n > 2f$ processes, where up to $f$ may crash.
Real-Time Clocks
Distributed Systems with RT Clocks

• Equip every processor $p$ with a local RT clock $C_p(t)$

• Small clock drift $\rho \rightarrow$ local clocks progress approximately as real-time, with clock rate $\in [1-\rho, 1+\rho]$

• End-to-end delay bounds $[\tau^-, \tau^+]$, a priori known
The Role of Real-Time

- Real-time clocks enable both:
  - Failure detection
  - Event ordering

- [Show later: Real-time clocks are not the only way …]
Failure Detection: Timeout using RT Clock

\[ \text{status} = \text{do_roundtrip}(q) \]
\[
\{ \text{send ping to } q \\
\text{TO} := C_p(t) + 5 \text{ seconds} \\
\text{wait until } C_p(t) = \text{TO} \\
\text{if pong did not arrive then} \\
\quad \text{return DEAD} \\
\text{else} \\
\quad \text{return ALIVE} \\
\}\]

\( p \) can reliably detect whether \( q \) has been alive recently, if
- the end-to-end delays are at most \( \tau^+ = 2.5 \text{ seconds} \)
- \( \tau^+ \) is known a priori [at coding time]
Event Ordering: Via Clock Synchronization

**Internal CS:**
- Precision $|C_p(t) - C_q(t)| \leq \pi$
- Progress like RT (small drift $\rho$)
- CS-Alg must periodically resynchronize

**External CS:**
- Accuracy $|C_p(t) - t| \leq \alpha$
- CS-Alg needs access to RT
- External CS $\rightarrow$ internal CS $\pi = 2\alpha$
Internal CS: Generate Periodic Resync Event

- **Via sync. clocks**
  
  + no message overhead
  
  - requires initial synchrony

- **Via message exchange [ST87]**

  - message overhead
  
  + no initial synchrony required

---

\[ C_p(t_1) = P \quad C_p(t_2) = 2P \quad C_p(t_3) = 3P \]

\[ C_q(t_1') = P \quad C_q(t_2') = 2P \quad C_q(t_3') = 3P \]
# Internal CS: Estimate Remote Clocks

- **One-way:**
  - 1 message only
  - Must know \( d \) and \( \varepsilon \)
    (and thus \( d_{\text{max}} = d + \varepsilon \))

- **Round-trip:**
  - 2 messages & larger error, BUT
  - Round-trip time \( U \) can be measured locally \( \rightarrow \) need to know \( d \) only [compute \( \varepsilon \)]
  - \( U \approx 2d \rightarrow \varepsilon \approx 0 \)
    “Probabilistic CS” [Cri89]

- **Estimate [at q]:**
  \[
  C_p(t_q) = kP + d + \varepsilon/2
  \]
  \( |\text{Error}| \leq \varepsilon/2 \)

- **Estimate [at p]:**
  \[
  C_q(t_p) = kP + 2d + \varepsilon
  \]
  \( |\text{Error}| \leq \varepsilon \)
Internal CS: FT Midpoint Algo [LWL88]

- A priori bounded $[\tau^-, \tau^+]$ allows to estimate all remote clocks
- Discard $f$ largest and $f$ smallest clock readings (could be faulty)
- Set local clock to midpoint of remaining interval

Before resync ...

\[ \pi' \leq \pi/2 \]

After resync ...

\[ p \]

\[ q \]
Internal CS in Biological Systems

• *Malaccae* Fireflies
  – Male fireflies emit light pulses ~ once per second
  – Swarm of fireflies eventually flash in synchrony

• Cardiac ganglion of lobster heart
  – Heart activation by synchronized firing of 4 interneurons
  – Ganglion controls activation frequency within some bounds, without losing synchrony
  – Evolution-optimized strategy: Fault-tolerance, self-stabilization, etc.
SS+FT Pulse Generation Alg. [DDP03]

- Pulse-coupled oscillators ("integrate-and-fire")
  - Time-decaying refractory function ~ own node’s sense of time
  - Time-decaying threshold level ~ perception of pulses from peers
  - Pulse is generated [+ refractory function reset] when refractory function hits threshold level

![Diagram showing pulse generation](chart.png)
SS+FT Clock Sync Algorithm [DDP03b]

- Allows to build linear-time self-stabilizing, Byzantine fault-tolerant clock sync algorithm, by
  - using synchronized pulses as a pacemaker
  - employing a self-stabilizing Byzantine agreement algorithm acting on clock values

- Also solves initial clock synchronization problem
External CS: Global Positioning System (GPS)

- 4 satellites required to determine $\chi = (x, y, z)$ and $\Delta$
- 1 satellite sufficient for $\Delta$ if $\chi$ is already known

Satellite clocks synchronized to USNO atomic master clock
GPS-Receiver solves system of equations

$$t_i + |\chi - s_i|/c + \Delta = T_i$$
Why are Synchronized Clocks Useful?

- Synchronized clocks allow to simulate communication-closed lock-step rounds via clock time [NT93]:

\[ C_p(t_1) = R \]
\[ C_p(t_2) = 2R \]
\[ C_p(t_3) = 3R \]

\[ C_q(t_1') = R \]
\[ C_q(t_2') = 2R \]
\[ C_q(t_3') = 3R \]

- Only requirement: \( R \geq \tau^+ + \pi \) holds!

- Lock-step rounds \( \longrightarrow \) perfect failure detection at end of rounds
Perfect FD $\rightarrow$ Lock-Step Round Round Simulation

- Attempt round simulation at $p$: Waiting for either
  - arrival of round message from $q$, or
  - $p$‘s instance of P suspects $q$

- Problem faced by $q$:
  - $msg_k$ not received in round $k$, although $p$ alive after round $k$
  - $q$ even receives $msg_{k+1}$ in round $k+1$ in this example
Using RT Clocks: Deficiencies

• Algorithms like do_roundtrip(.) have system-dependent time values (unit „seconds“) in their code / variables → not easily portable to e.g. faster hardware

• Fail-operational systems might tolerate occasional loss of timeliness properties – but never of safety properties

• Unfortunately:
  ❖ Safety properties like agreement typically rely on the reliable operation of do_roundtrip(.) and similar primitives
  ❖ End-to-end delay bounds $\tau^+$ that always hold are difficult to determine in real systems

➢ Try to relax timing assumptions in ParSync models …
Food for Thoughts

(1) Consider the fault-tolerant midpoint algorithm for computing the clock corrections $u=\text{FTM}(U)$ resp. $v=\text{FTM}(V)$ for the given vectors $U=(u_1,\ldots,u_n)$ resp. $V=(v_1,\ldots,v_n)$ of $n \geq 3f + 1$ clock readings (ordered by process ids). Assume that at least $n - f$ of the pairs $(u_i, v_i)$ satisfy

- $|u_i - v_i| \leq \varepsilon$
- $|u_i - u_j| \leq \pi$
- $|v_i - v_j| \leq \pi$

Prove that $|u - v| \leq \pi/2 + 2\varepsilon$. 
Partially Synchronous Models
Recall: Synchronous Model

- "The" classic model
  - Transmission delay bound $\tau^+$
  - Computing step time bound $\mu^+$
  - Bounded-drift local clocks available

- Allows (Byzantine-tolerant) implementation of
  - Internal clock synchronization
  - Lock-step rounds
  - etc.
The Timed Asynchronous Model

• Cristian & Fetzer [CF99]:
  – Alternating bad and good periods:
    • Transmission delay bound $\tau^+$
    • Computing step time bound $\mu^+$
  – Reliable bounded-drift local RT clocks available
  – Local clocks allow to detect good/bad periods $\Rightarrow$ TA algorithms are always safe and live in good periods

• TA algorithms allow to implement (non-Byzantine) fail-aware services, including eventual lock-step rounds
Classic Partially Synchronous Models (I)

• „The“ classic ParSync models
  Dolev, Dwork & Stockmeyer [DDS87]
  Dwork, Lynch & Stockmeyer [DLS88]
  Attiya, Dwork, Lynch & Stockmeyer [ADLS94]

• Semi-synchronous model by Ponzio & Strong [PS92]

• Common system parameters:
  – Bounded processor speed ratio $\Phi = \mu^+ / \mu^-$
  – Transmission delay bound $\Delta$

• Archimedean model by Vitanyi [Vit84]
  – Bounded speed ratio $S = \tau^+ / \mu^-$
Classic Partially Synchronous Models (II)

Processes can **locally time-out** messages:

- **The classic ParSync models** [DDS87, DLS88] and [ADLS94] assume
  - \( \Delta \) given in multiples of (unknown) minimal computing step time \( \mu^- \) [hence \( \tau^+ = \Delta \cdot \mu^- \) real-time seconds]
  - spin loop counting \( f(\Phi, \Delta) \) steps allows to time-out messages [implements local clock with real-time rate \( \in [1/\Phi, 1] \)]

- **Archimedean model** [Vit84] also allows to time-out messages via spin-loop for \( S \) steps

- **Semi-synchronous model** [PS92] assumes
  - \( \Delta = \tau^+ \) given in real-time seconds
  - bounded-drift local RT clocks available for timing-out messages
Classic Partially Synchronous Models (III)

Variants of ParSync models: System parameters ($\Delta$, $\Phi$)

1. known and hold from the beginning

2. known and hold from unknown global stabilization time (GST) on

3. unknown and hold from the beginning / from GST on:
   Learn ($\Delta$, $\Phi$), by continuously increasing estimate values
Time-Free Message-Timeout in ParSync ?

• Implementation of do_roundtrip(p) in the ParSync models of [DLS88] or [Vit85]:

```plaintext
{  send ping to p
    for i=1 to x do no-op       /* x=f(Δ, Φ) resp. x=f(s) is
                               dimensionless! */
    if pong did not arrive then
        return DEAD
    else
        return ALIVE
}
```

• But: No obvious correlation between processor step times and message delays \(\Rightarrow\) not really time-free …
The $\Theta$/ABC-Model

For classic ParSync models:
- Timing assumptions are primarily used for ordering events
- Actual duration ($D$) irrelevant
- Is it possible to define a time-free ParSync model based on event ordering in the first place?

For example ParSync models:
- only less than $\Theta$ roundtrips can occur during any single round-trip
- $\Theta = 5$

\[
\begin{align*}
\text{status} &= \text{do\_roundtrip}(q) \\
&\begin{cases}
\text{send ping to } q \\
\text{for } i=1 \text{ to } \Theta \text{ do} \\
\text{begin} \\
\text{send delay\_ping}(i) \text{ to } r \\
\text{wait for delay\_pong}(i) \text{ from } r \\
\text{end} \end{cases} \\
\text{if pong did not arrive then} \quad \text{return DEAD} \\
\text{else} \quad \text{return ALIVE}
\end{align*}
\]
The $\Theta$-Model: Bounded E-t-E Delay Ratio

Widder & Schmid [WS09]

- End-to-end delays of all messages in transit at $t$
  - minimum $\tau^-(t)$
  - maximum $\tau^+(t)$

- $\tau^+(t)$ and $\tau^-(t)$ may vary arbitrarily with time, but:

- Ratio $\tau^+(t)/\tau^-(t)$ bounded by [known or even unknown] system parameter $\Theta$
Byzantine FT Clock Sync in the $\Theta$-Model

For $n \geq 3f + 1$ with up to $f$ Byz. failures:

- Suppose $p$ sends $\text{tick}(C+1)$ at time $t$
- Then, $q$ also sends $\text{tick}(C+1)$ by time $t + 2\tau^+ - \tau^-$

+ Fastest tick-frequency of any $p$: $1/\tau^-$

$\Rightarrow$ Clock ticks occur approximately synchronously, with precision $\pi(\Theta)$

**On init**

- $\rightarrow$ send $\text{tick}(0)$ to all; $C := 0$;

**If got $\text{tick}(l)$ from $f+1$ nodes and $l > C$**

- $\rightarrow$ send $\text{tick}(C+1), \ldots, \text{tick}(l)$ to all;
  - $C := l$;

**If got $\text{tick}(C)$ from $2f+1$ nodes**

- $\rightarrow$ send $\text{tick}(C+1)$ to all;
  - $C := C+1$;

---

U. Schmid
Correlation → Coverage Expansion

- Given some bound $\tau^+$ and $\tau^-$ assumed during system design (as used in synchronous systems), compute $\Theta = \tau^+ / \tau^-$

- Unanticipated overload: $\tau^+(t) > \tau^+$ — if $\tau^+(t) \leq \Theta \tau^-(t)$, however,

Note:
- $\tau^+(t) = \tau^+ + \alpha(t)$
- $\tau^-(t) = \tau^- + \frac{\alpha(t)}{\Theta}$
  sufficient for $\Theta$ to hold!
Shortcomings $\Theta$-Model

• Correlation between slow and fast messages need not exist for all messages
  – Some very fast messages [even $\tau^- = 0$] may be in transit somewhere in the system during a slow message
  – Correlation and hence coverage expansion does not exist in such cases

• Need a more relaxed definition of the relation between slow and fast messages
  – All that is actually needed is to constrain the number of fast messages during a slow one
  – No need for a correlation of unrelated messages, and at every point in time $t$
The Asynchronous Bounded Cycle Model

Robinson & Schmid [RS08]

- The ABC Model just bounds the ratio of the number of forward and backward-oriented messages in cycles

- Example: $\Theta = 4.5$
- 2 consecutive "slow" messages
- Cycle with 9 enclosed "fast" messages
- No larger cycles allowed

- No implicit or explicit reference to real-time
  - Messages with $\tau^{-}(t) = 0$ allowed
  - No need to relate independent messages in the system
  - We proved: Any $\Theta$-algorithm works correctly in the ABC model
Partial Order of ParSync Models

• DLS … [DLS88] with known $\Delta, \Phi$
• $\Theta$ … ABC/$\Theta$-Model with known $\Theta$
• DLS$^u$ … [DLS88] with unknown $\Delta, \Phi$
• $\Theta^u$ … ABC/$\Theta$-Model with unknown $\Theta$
• FLP … asynchronous FLP-Model
Even Weaker ParSync Models?

• All the ParSync Models seen so far allow to build
  – lock-step rounds, or at least
  – eventual lock-step rounds

• Solving consensus is easy here.

• We know that lock-step rounds are stronger than failure detectors that are sufficient for solving consensus:
  – Perfect failure detector P
  – Leader oracle $\Omega$

• Are there weaker ParSync models where only such FDs can be implemented?
Weaker Partially Synchronous Models
Finite Average Roundtrip-Time Model (I)

Fetzer, Schmid and Süskraut [FSS04]

- Asynchronous system with crash failures
- Unknown lower bound $\mu^-$ for computing step time
- Unknown average round-trip time bounds

\[ \lim_{n \to \infty} \frac{1}{n} \sum_{k=1}^{n} RTT(k) < \infty \]

- $RTT(k)$ and hence $\tau^+$ unbounded, yet
- Average after $n$ „Epochs“ is

\[
\frac{n(n+1)}{n(n+1)-(n-1)n/2} = 2 \cdot \frac{n^2 + n}{n^2 + 3n} < \infty
\]
Finite Average Roundtrip-Time Model (II)

• The FAR model assumptions
  – do not allow to implement lock-step rounds
  – do allow to implement the eventually perfect FD P
  – can solve consensus if $n > 2f$

• Key ideas for P implementation:
  – Implement weak local clock [via spin-loop] for timing-out messages
  – Time-out roundtrips using adaptive timeout value TV
    • If fast RT occurs [before TO]: Increase TV, to prepare for future slow RTs
    • If slow RT occurs [after TO]: (Could) decrease TV, since fast RTs must eventually follow due to finite average RTT
Weak Timely Link Models (I)

Aguilera, Delporte, Fauconnier, Toueg [ADFT04], Hutle, Malkhi, Schmid, Zhou [HMSZ09]:

• Partially synchronous processors (Φ) with crash failures
• Almost all communication asynchronous, except:
• At least one process $p$ must be an $\Diamond f$-source:
  – After some (unknown) time, $p$ has timely links to at least $f$ neighbors
    [No message sent at time $t$ is processed after $t+\tau^+$ (unknown)]
  – Note: A link to a crashed process is timely per definition!
• Allows to implement $\Omega$, and hence solving consensus for $n > 2f$
• An $\Diamond f$-1-source is provably not sufficient
• Currently weakest WTL model [HMSZ09]: A moving $\Diamond f$-source, where the $f$ timely links can change with time
Weak Timely Link Models (II)

Ω implementation: Every process

- periodically broadcasts heartbeat message (HB)
- times-out HBs of all neighbors
  - using weak local clock [implemented via step counting in spin-loop]
  - timeout value increased on every TO [= no HB received before expiration]
- broadcasts accusation message acmsg(q) on every TO for q’s HB
- if n-f acmsg(q) are received, then increment acc_count[q]
- Ω-output: q with min. acc_count[q]

➢ All processes accuse crashed r ➔ acc_count[r] continuously grows
➢ 5+1 processes never accuse p ➔ incrementing acc_count[p] stops
Even Weaker Models (I)

- Investigate models for weaker problems than consensus

- Candidate of choice: $k$-set agreement [Cha93]:
  - Input values from finite domain $V$ with $|V| > k$
  - Processes must decide on at most $k$ different output values system-wide

- Well-known properties:
  - Weakening of consensus (= 1-set agreement)
  - Requires $\lceil f/k \rceil + 1$ rounds in synchronous systems with up to $f$ crashes
  - Impossible in asynchronous systems if $f \geq k$ crashes
Even Weaker Models (II)

- $k$-set agreement allows to further explore the synchronous/asynchronous solvability border
- There are models where
  - $k$-1-set agreement (hence consensus) is impossible
  - $k$-set agreement is possible
- Two major directions of research:
  - Failure detectors
  - ParSync models
Dynamic Networks
Synchronous Distributed Computations with Time-Varying Communication Graphs

Joining/leaving nodes

Appearing/disappearing links

Network partitioning

Round $k$  Round $k+1$  Round $k+2$
Study Agreement Problems

• **Consensus**
  • Processes have local input value and local decision value (initially undefined)
  • Agreement: Processes must decide on a single common output value system-wide, within some finite termination time

• **Weaker problem: $k$-Set agreement**
  – $k$-Agreement: Processes must decide on at most $k$ different output values system-wide
  – Relaxation of consensus agreement property

• **Weaker problem: Approximate agreement**
  – Processes must decide on values that are within $\varepsilon$ of each other
  – Relaxation of consensus agreement property
Possible Applications

- **Approximate agreement:** Clock synchronization

- **Gracefully degrading k-set agreement:** Transmission schedule negotiation

\[ k = 2 \]
Message Adversary-based Modeling

Message adversary (MA) chooses sequence of communication graphs

Strong MA

Weak MA

if (..) somethinguseful else somethingsmoreuseful

Impossibleity

Algorithm

U. Schmid

182.703 PRDC
Network Assumptions

• Adversarial model:
  – Message adversary chooses communication graph sequence
  – Restricted by network assumptions

• Strong network assumption (e.g. always strongly connected)
  + Solution algorithms simple
  – Assumption coverage in real systems small

• Weak network assumption
  – Complex, expensive algorithms (if existing at all)
  + Assumption coverage in real systems large
Consensus Solvability/Impossibility

[BR12] [SW89]
Consensus Solvability/Impossibility

[NSW19], using point-set topology

Adversary weakness

Algorithmic complexity

Impossibility/solvability border!
The End
(Part 3)
References


References

References


References