Contention-Related Crash Failures

Anaïs Durand[*], Michel Raynal[*,†], and Gadi Taubenfeld[‡]

[*] IRISA, Université de Rennes, France
[†] Polytechnic University, Hong Kong
[‡] Interdisciplinary Center, Herzliya, Israel

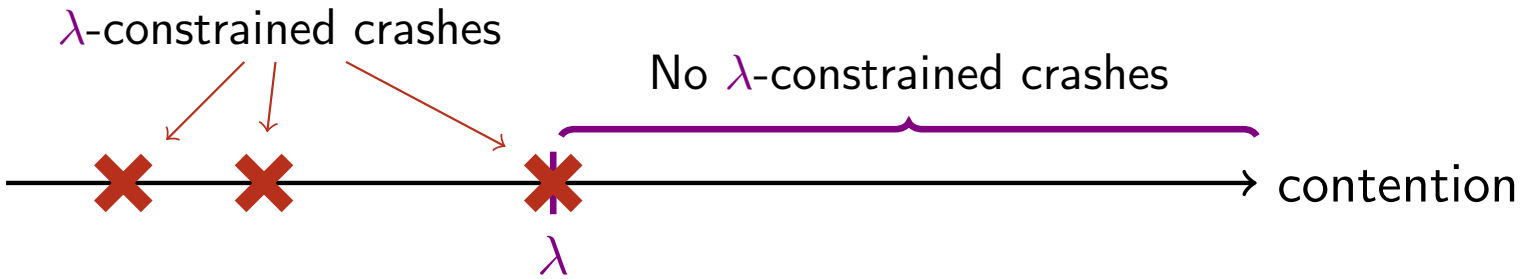
October 1st, 2018

Computational Model

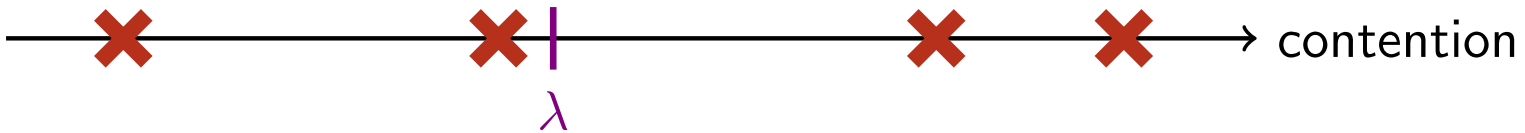
- Asynchronous deterministic system
- n processes $p_1, \ldots, p_n$
- Atomic read/write registers
- Process crashes
- Participation required

Contention-Related Crash Failures [Taubenfeld,18]

- **Contention** = # processes that accessed a shared register
- λ = predefined contention threshold
- 2 kinds of crash failures:
  - **λ-constrained** crash failures:



  - **"any-time"** crash failures:



Contention-Related *vs.* Any-Time Crash Failures

- **Consensus:**
  - [Fischer *et al.*, 85]: **Impossible** with **one any-time** crash failure.
  - [Taubenfeld, 18]: Algorithm that tolerates **one** $(n-1)$**-constrained** crash failure for $n > 1$.

- k**-Set Agreement,** $1 \leq k < n$**:**
  - [Borowsky, Gafni, 93]: **Impossible** with k **any-time** crash failures.
  - [Taubenfeld, 18]: Algorithm that tolerates $\ell + k - 2$ $(n-\ell)$**-constrained** crash failures for $\ell \geq 1$ and $n \geq 2\ell + k - 2$.

Motivation

Consider a problem P that can be solved with t any-time crash failures.

Given λ, can P be solved with both

t_1 **λ-constrained**

and

t_2 **any-time**

crash failures, with $t_1 + t_2 > t$?

We consider here: **k-set agreement** (for $k \geq 2$) and **renaming**

k-Set Agreement

k-Set Agreement [Chaudhuri,90]

Definition

- One-shot object
- Operation *propose*(v): propose value v and return a decided value
- Properties:
  - **Validity:** decided value = proposed value
  - **Agreement:** $\leq k$ decided values
  - **Termination:** every correct process decides

k-Set Agreement Algorithm: Properties

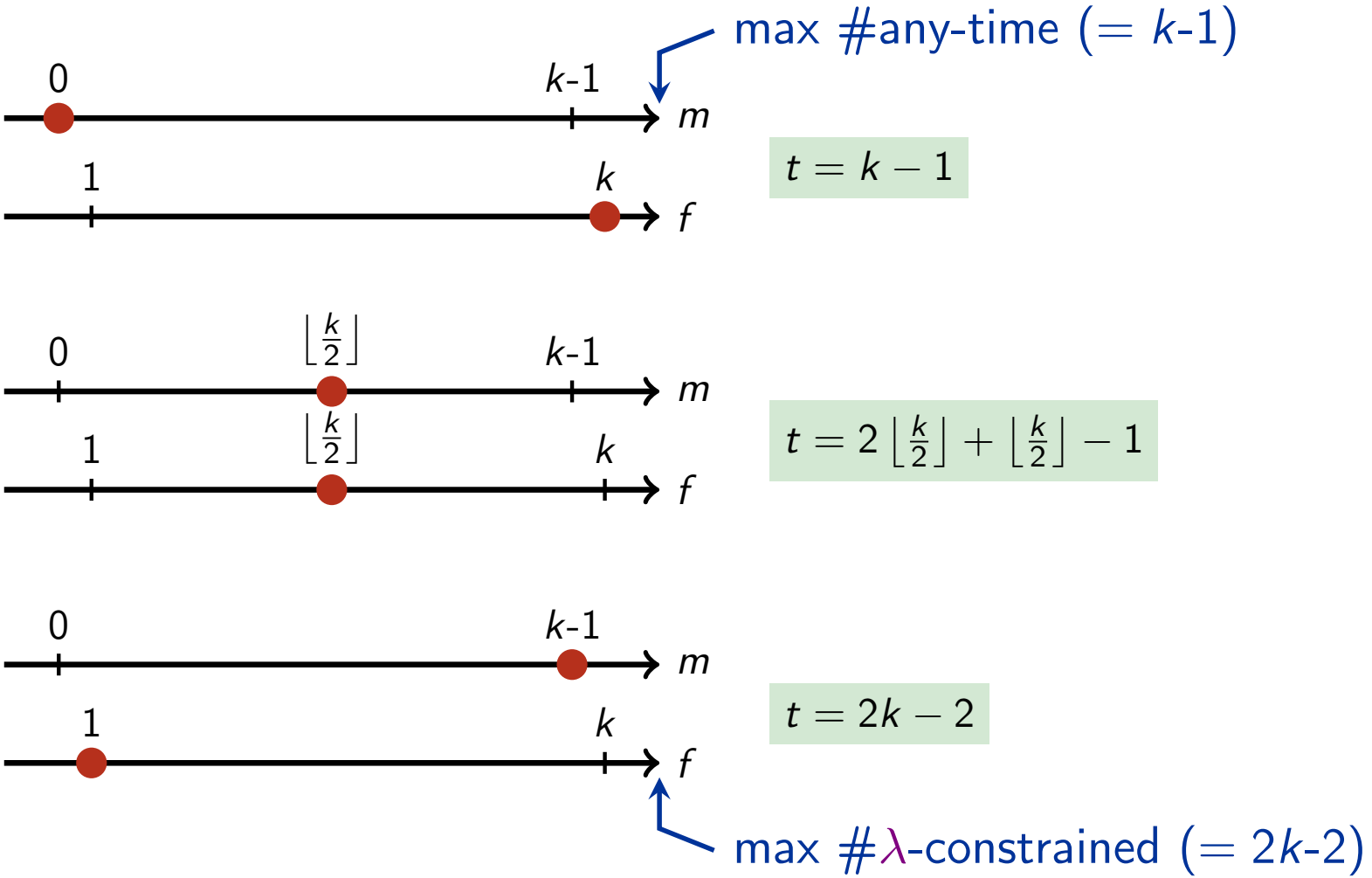
- $\lambda = n - k$
- $k \geq 2$
- $k = m + f$, $m \geq 0$, $f \geq 1$

total # of faults	$t = 2m + f - 1 = k + m - 1$
λ-constrained crashes	$2m$
any-time crashes	$f - 1$

[Borowsky, Gafni, 93]: Impossible with k any-time crash failures.

k-Set Agreement: Parameters

Parameters f and m allow the user to **tune** the proportion of each type of crash failures.



k-Set Agreement: Shared Registers (1/2)

- DEC: atomic register, initially $\perp$
- $PART[1 \dots n]$: snapshot object, initially $[\texttt{down}, \dots, \texttt{down}]$

  - Atomic (linearizable) operations *write*() and *snapshot*()
  - $\approx$ array of single-writer multi-reader atomic register $PART[1 \dots n]$ such that:
    - p_i invokes $write(v)$ = writes v into $PART[i]$
    - p_i invokes *snapshot*() = obtains the value of the array $PART[1 \dots n]$ as if it read simultaneously and instantaneously all its entries

k-Set Agreement: Shared Registers (2/2)

- *MUTEX*[1]: one-shot deadlock-free f-mutex
- *MUTEX*[2]: one-shot deadlock-free m-mutex
  - Operations *acquire*() and *release*() (invoked at most once)
  - Properties:
    - Mutual exclusion: $\leq m$ processes simultaneously in critical section
    - Deadlock-freedom: if $< m$ processes crashes, then ≥ 1 process invoking *acquire*() terminates its invocation

k-Set Agreement Algorithm (1/2)

operation $propose(in_i)$ **is**

(1) $PART.write(\texttt{up})$; % *signal participation*

k-Set Agreement Algorithm (1/2)

operation $propose(in_i)$ **is**

(1) $PART.write(\mathtt{up})$; % *signal participation*

(2) **repeat**
(3) $part_i := PART.snapshot()$; % *wait for* $n - t$
(4) $count_i := |\{x \text{ such that } part_i[x] = \mathtt{up}\}|$; % *participants*
(5) **until** $count_i \geq n - t$ **end repeat**;

k-Set Agreement Algorithm (1/2)

operation $propose(in_i)$ **is**

(1) $PART.write(\mathtt{up})$; % *signal participation*

(2) **repeat**
(3) $\quad part_i := PART.snapshot()$; % *wait for* $n - t$
(4) $\quad count_i := |\{x$ such that $part_i[x] = \mathtt{up}\}|$; % *participants*
(5) **until** $count_i \geq n - t$ **end repeat**;

(6) **if** $count_i \leq n - k$ **then** % *split processes into groups*
(7) $\quad group_i := 2$; % $\rightsquigarrow$ *MUTEX*[2] *(m-mutex)*
(8) **else**
(9) $\quad group_i := 1$; % $\rightsquigarrow$ *MUTEX*[1] *(f-mutex)*
(10) **end if**

k-Set Agreement Algorithm (1/2)

operation $propose(in_i)$ **is**

(1) $PART.write(\mathtt{up})$; % *signal participation*

(2) **repeat**
(3) $\quad part_i := PART.snapshot()$; % *wait for* $n - t$
(4) $\quad count_i := |\{x$ such that $part_i[x] = \mathtt{up}\}|$; % *participants*
(5) **until** $count_i \geq n - t$ **end repeat**;

(6) **if** $count_i \leq n - k$ **then** % *split processes into groups*
(7) $\quad group_i := 2$; % $\rightsquigarrow$ *MUTEX*[2] *(m-mutex)*
(8) **else**
(9) $\quad group_i := 1$; % $\rightsquigarrow$ *MUTEX*[1] *(f-mutex)*
(10) **end if**

(11) launch in // the threads T_1 and T_2;

k-Set Agreement Algorithm (2/2)

```
thread T_1 is
(1)   loop forever
(2)       if DEC ≠ ⊥ then
(3)           return(DEC);
(4)       end if;
(5)   end loop;
```

% wait for a decided value

k-Set Agreement Algorithm (2/2)

```
thread T1 is                                   % wait for a decided value
(1)   loop forever
(2)       if DEC ≠ ⊥ then
(3)           return(DEC);
(4)       end if;
(5)   end loop;
```

```
thread T2 is                                   % decide a value if enters its CS
(6)   if group_i = 1 ∨ m > 0 then
(7)       MUTEX[group_i].acquire();
(8)           if DEC = ⊥ then
(9)               DEC := in_i;
(10)          end if
(11)      MUTEX[group_i].release();
(12)      return(DEC);
(13)  end if;
```

k-Set Agreement Algorithm: Validity & Agreement

```
thread T1 is
(1)   loop forever
(2)       if DEC ≠ ⊥ then
(3)           return(DEC);
(4)       end if;
(5)   end loop;

thread T2 is
(6)   if group_i = 1 ∨ m > 0 then
(7)       MUTEX[group_i].acquire();
(8)           if DEC = ⊥ then
(9)               DEC := in_i;
(10)          end if
(11)      MUTEX[group_i].release();
(12)      return(DEC);
(13)  end if;
```

a Decided value = DEC

k-Set Agreement Algorithm: Validity & Agreement

```
thread T1 is
(1)   loop forever
(2)       if DEC ≠ ⊥ then
(3)           return(DEC);
(4)       end if;
(5)   end loop;

thread T2 is
(6)   if group_i = 1 ∨ m > 0 then
(7)       MUTEX[group_i].acquire();
(8)           if DEC = ⊥ then
(9)               DEC := in_i;
(10)          end if
(11)      MUTEX[group_i].release();
(12)      return(DEC);
(13)  end if;
```

- **a** Decided value = DEC
- **b** DEC assigned to **proposed** values in_i in CS

k-Set Agreement Algorithm: Validity & Agreement

```
thread T_1 is
(1)   loop forever
(2)       if DEC ≠ ⊥ then
(3)           return(DEC);
(4)       end if;
(5)   end loop;

thread T_2 is
(6)   if group_i = 1 ∨ m > 0 then
(7)       MUTEX[group_i].acquire();
(8)           if DEC = ⊥ then
(9)               DEC := in_i;
(10)          end if
(11)      MUTEX[group_i].release();
(12)      return(DEC);
(13)  end if;
```

- **a** Decided value = DEC
- **b** DEC assigned to **proposed** values in_i in CS
- **c** $MUTEX[1] \rightsquigarrow \leq f \neq$ values
  $MUTEX[2] \rightsquigarrow \leq m \neq$ values

  $\Rightarrow \leq f + m = k$ **decided values**

k-Set Agreement Algorithm: Termination

```
(1)  PART.write(up);
(2)  repeat
(3)      part_i := PART.snapshot();
(4)      count_i := |{x such that part_i[x] = up}|;
(5)  until count_i ≥ n − t end repeat;
```

a $\leq t$ crashes + participation required
$\rightsquigarrow$ eventually $count_i \geq n - t$ at every correct process p_i

k-Set Agreement Algorithm: Termination

```
(6)   if count_i ≤ n − k then
(7)       group_i := 2;
(8)   else
(9)       group_i := 1;
(10)  end if
```

a $\leq t$ crashes + participation required
$\rightsquigarrow$ eventually $count_i \geq n - t$ at every correct process p_i

b $group_i = 1 \Rightarrow count_i > n - k = \lambda$
$\rightsquigarrow$ no λ-constrained crashes among participants of group 1
$\rightsquigarrow \leq f - 1$ crashes in f-mutex $MUTEX$[1]

k-Set Agreement Algorithm: Termination

a $\leq t$ crashes + participation required
$\rightsquigarrow$ eventually $count_i \geq n - t$ at every correct process p_i

b $group_i = 1 \Rightarrow count_i > n - k = \lambda$
$\rightsquigarrow$ no λ-constrained crashes among participants of group 1
$\rightsquigarrow \leq f - 1$ crashes in f-mutex $MUTEX$ [1]

c **If** ≥ 1 correct process $\in$ group 1 $\rightsquigarrow \geq 1$ of them decides
otherwise (some maths) $\rightsquigarrow \leq m - 1$ crashes in m-mutex $MUTEX$ [2]
& ≥ 1 correct process in group 2 $\rightsquigarrow \geq 1$ of them decides

Renaming

Definition

- Initial name: id_i
- New name space: $\{1 \dots M\}$
- Operation *rename*(id_i): return a new name
- Properties:
  - **Validity:** new name $\in \{1 \dots M\}$
  - **Agreement:** no 2 same new names
  - **Termination:** invokation of *rename*() by a correct process terminates

Renaming Algorithm: Properties

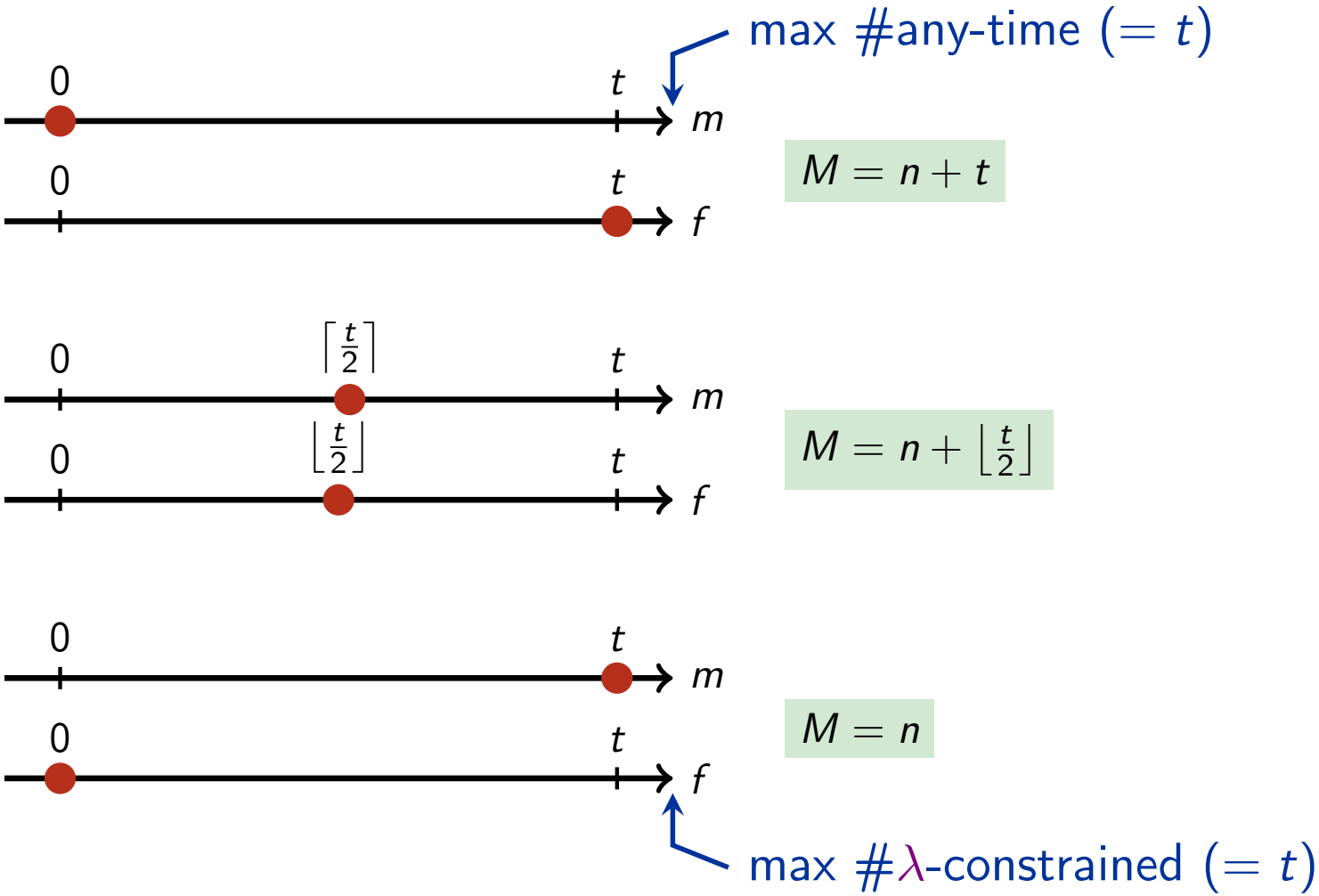
- $M = n + f$
- $\lambda = n - t - 1$
- $t = m + f$, $m \geq 0$, $f \geq 0$

total # of faults	$t = m + f$
λ-constrained crashes	m
any-time crashes	f

[Herlihy, Shavit, 93]: Impossible with $f + 1$ any-time crash failures.

Renaming Algorithm: Parameters

Parameters f and m allow the user to **tune** the proportion of each type of crash failures and the size of the new name space.



Renaming Algorithm: Shared Registers

- $PART[1 \dots n]$: snapshot object, initially $[\texttt{down}, \dots, \texttt{down}]$

- $RENAMING_f$: $(n+f)$-renaming object that:
  - tolerates $\leq f$ any-time crash failures
  - does not require participation

  e.g. [Attiya, Welch, 04]

Renaming Algorithm

operation *rename*(id_i) **is**

```
(1)  PART.write(up);                                  % signal participation
(2)  repeat
(3)      part_i := PART.snapshot();                   % wait for n − t
(4)      count_i := |{x such that part_i[x] = up}|;   % participants
(5)  until count_i ≥ n − t end repeat;
```

Renaming Algorithm

operation $rename(id_i)$ **is**

```
(1)  PART.write(up);                                  % signal participation
(2)  repeat
(3)      part_i := PART.snapshot();                   % wait for n − t
(4)      count_i := |{x such that part_i[x] = up}|;   % participants
(5)  until count_i ≥ n − t end repeat;

(6)   newName_i := RENAMING_f.rename(id_i);           % get new name
(7)   return(newName_i);
```

Renaming Algorithm: Proof

```
(1)  PART.write(up);
(2)  repeat
(3)      part_i := PART.snapshot();
(4)      count_i := |{x such that part_i[x] = up}|;
(5)  until count_i ≥ n − t end repeat;
```

a $\leq t$ crashes + participation required
$\rightsquigarrow$ eventually $count_i \geq n - t$ at every correct process p_i

Renaming Algorithm: Proof

(1) $PART.write(\mathtt{up})$;
(2) **repeat**
(3) $\quad part_i := PART.snapshot()$;
(4) $\quad count_i := |\{x \text{ such that } part_i[x] = \mathtt{up}\}|$;
(5) **until** $count_i \geq n - t$ **end repeat**;

a $\leq t$ crashes + participation required
$\rightsquigarrow$ eventually $count_i \geq n - t$ at every correct process p_i

b $n - t > \lambda \rightsquigarrow$ no λ-constrained crashes in $RENAMING_f$
$\rightsquigarrow \leq f$ crashes in $RENAMING_f$

Renaming Algorithm: Proof

(1) $PART.write(\mathtt{up})$;
(2) **repeat**
(3) $\quad part_i := PART.snapshot()$;
(4) $\quad count_i := |\{x \text{ such that } part_i[x] = \mathtt{up}\}|$;
(5) **until** $count_i \geq n - t$ **end repeat**;

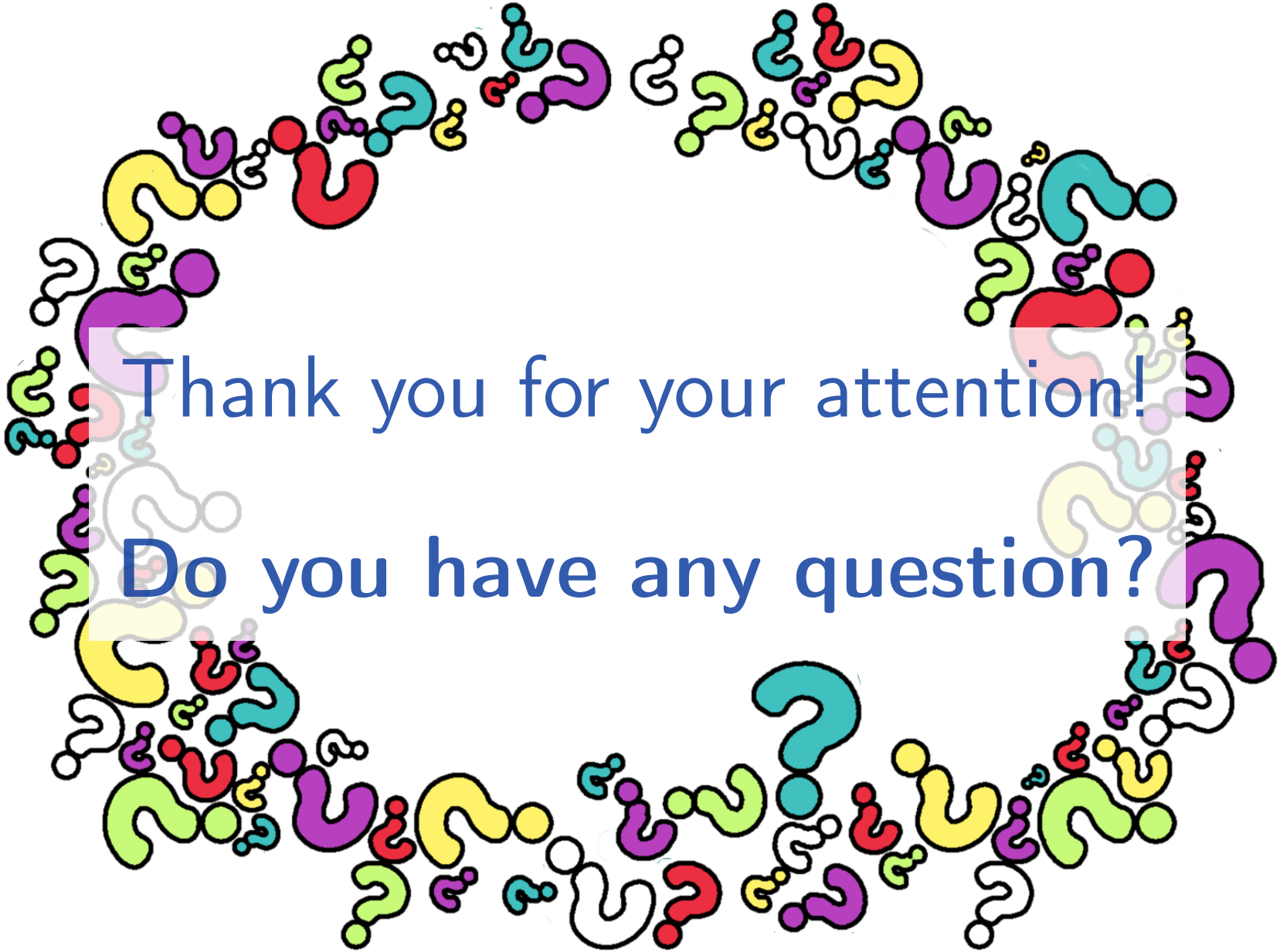
a $\leq t$ crashes + participation required
$\rightsquigarrow$ eventually $count_i \geq n - t$ at every correct process p_i

b $n - t > \lambda \rightsquigarrow$ no λ-constrained crashes in $RENAMING_f$
$\rightsquigarrow \leq f$ crashes in $RENAMING_f$

c participation not required for $RENAMING_f$ + properties of $RENAMING_f$
$\rightsquigarrow$ validity, agreement, & termination

Conclusion

- Notion of **contention-related** crash failures
- Allows to circumvent impossibility results
- Future work:
  - Tight bounds?
  - General algorithm for k-set agreement, $k \geq 1$.



Thank you for your attention!

Do you have any question?

Generalization to One-Shot Concurrent Objects

Transform OB = one-shot object tolerating $< X$ any-time crashes, participation not required

- $\lambda = n - t - 1$
- $t = m + f$, $m \geq 0$, $0 \leq f \leq X$

total # of faults	$t = m + f$
λ-constrained crashes	m
any-time crashes	$f \leq X$

operation $op(in_i)$ **is**

(1) $PART.write(\mathrm{up})$;
(2) **repeat**
(3) $\quad part_i := PART.snapshot()$;
(4) $\quad count_i := |\{x$ such that $part_i[x] = \mathrm{up}\}|$;
(5) **until** $count_i \geq n - t$ **end repeat**;

(6) $res_i := OB.op(in_i)$;
(7) **return**(res_i);