ReStore — Redundant Storage

- in-memory
- replicated across the PEs with user definable replication level

Data distribution

We assume nothing about the load-balane of the programm which uses the ReStore library. After a failure of a PE, the working copies of the data might be distributed completely independent of how they were distributed before the first failure. To simplify the logic which enables each PE to know where it can fetch the data it needs from, we decided not to use the working copies as an extra ~~copy~~ replica of the data during restoration.

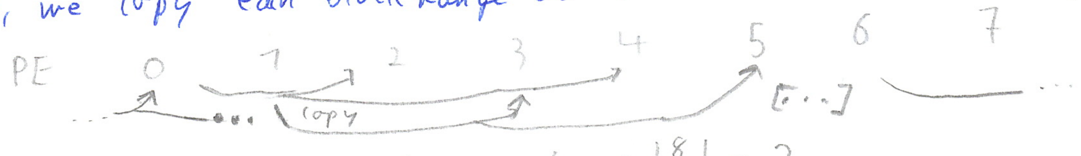
How we keep the data
- We assume, that the data can be split into B independent blocks e.g. sites in RAxML — we avoid storing the block's ids esplicitly, then can be arbitrarly small without wasting space
- These blocks should be about equal in size and there should be way more blocks than PEs (number of PEs: P)
- The user ~~defines~~ sets a replikation level K
- ~~Ha~~ We divide the B blocks in P block ranges. Blocks with an adjacent id are grouped together → continuous read. needs less messages

Example P = 8 B = 66

PE	⓪	①	②	③	...
Blocks	0 1 2 3 4 5 6 7 8 9	10 11 ··· 18	19 ··· 26	27 ··· 34	...
	9 blocks	9 blocks	8 blocks	8 blocks	8 blocks
Block Range	0	1	2	3	...

- Next, we copy each block range to k-1 other PEs

k=3
P=8



shift = $\lfloor \frac{P}{k} \rfloor$ in this example $s = \lfloor \frac{8}{3} \rfloor = 2$

this is the maximum distance the replicas can have
→ maximum robustness if we assume that ranks which are close together have a higher probability of failing together (same machine, rack, island)

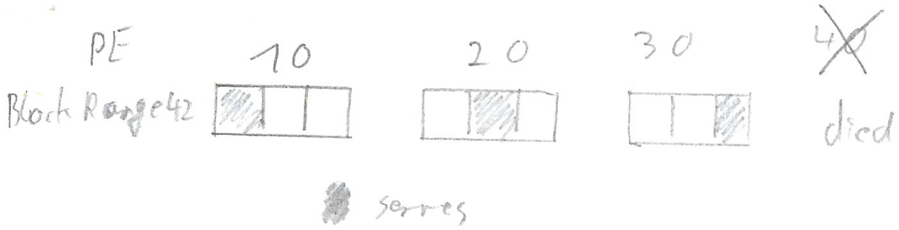
- This data distribution allows each PE to determine where each copy of each data block resides at all times while only using $O(\#failures)$ memory to store the failed PEs

After a failure, the load balancer is run. It redistributes the work in a unpredictable (i.e. we make no assumptions about it) fashion.

Each PE then knows, which blocks it needs. and where these blocks are stored

Each PE knows which blocks it can serve and how much other PEs can save the same block. These PEs can then evenly divide serving a block range. e.g.:

Block Range 10 was stored on PEs 10 20 30 and 40



→ We can perform ~~a~~ Sparse All to All to send each PE the blocks it needs.

(We need a second Sparse AllToAll to request the blocks if they are only known at the receiving side.)

Sparse All to All

1. Push all messages into the network using a non-blocking but synchronous[1] send
2. while not all..... messages I sent are received   MPI_Testall
   receive message sent to me
3. Enter non blocking barrier „everything I sent was received"
4. while not everyone is in that barrier
   receive message sent to me

[1] Testall will succeed only when the message has been received

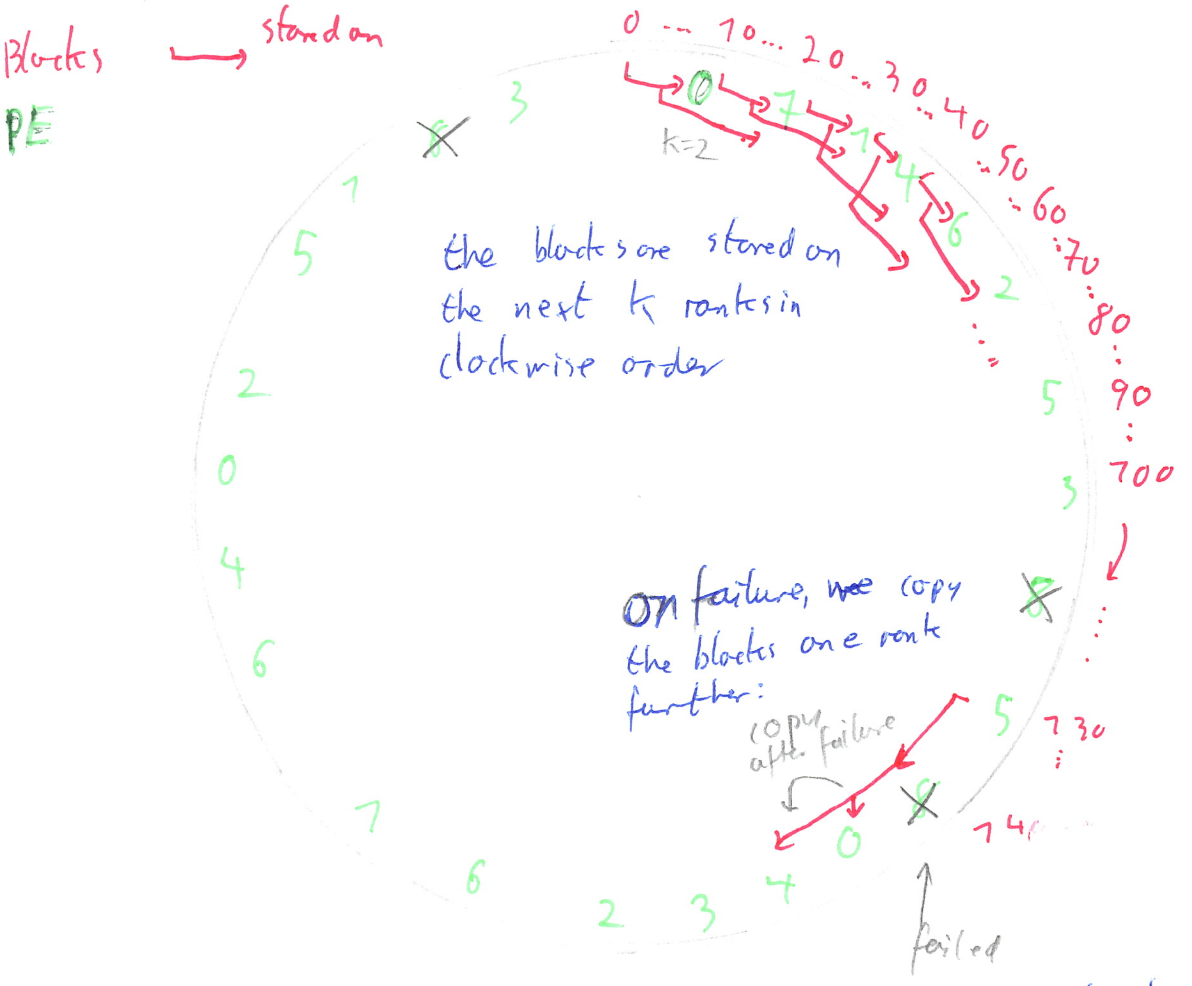
Possible redistribution of replicas after a rank failure

Use consisten hashing

↳ Map the blocks onto the unit circle

↳ Map each PE multiple times$^{(g)}$ onto the unit circle

Traditionally this is done randomly, we think we can find a deterministic solution.



the blocks are stored on the next k ranks in clockwise order

On failure, we copy the blocks one rank further:

- On failure, only the lost replicas have to be restored no additional copies are send over the network
- ... The larger g, the smaller the imbalance after a failure