Cooling storage hotspots in the data centre

Toby Moncaster, George Parisis, Anil Madhavapeddy, Jon Crowcroft (first.last@cl.cam.ac.uk)

Data Centres: Where the 'net Things Are

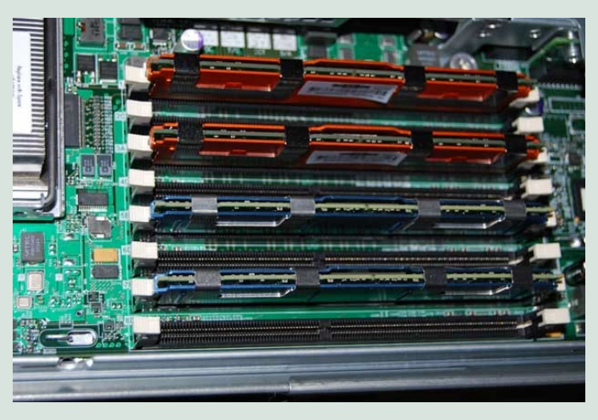
It's all about the numbers!

- **Exabytes of data**
- **Millions of processors**
- **Hundreds of thousands of servers**
- **Tens of MW of power**

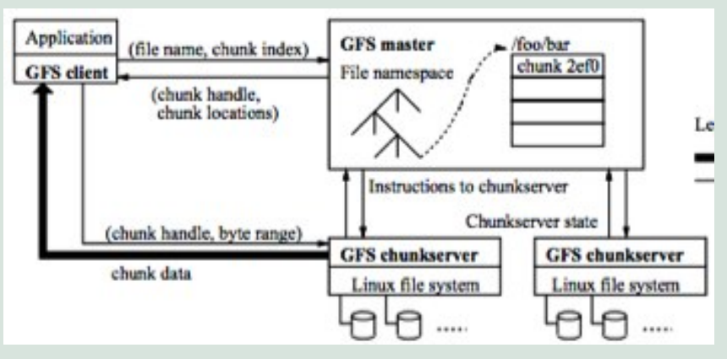
Used for web services like Google, Social networking (Facebook), Cloud services (Amazon EC2) and video streaming (NetFlix).

Storage, Storage, Storage!

All that data needs to live somewhere:



- **in-memory storage.** Keep it all in RAM. Fast, but volatile. Also requires complex coordination.



- **local-attached storage.** Store on local disc with central metadata. Slower, less volatile, but still needs coordination. Typical examples include Flat Datacenter Storage which is the basis for Trevi

- **network-attached storage.** Central file server(s) appear as local storage on the actual server. Uses significant network bandwidth. Slower. Allows easy replication. Typical example is NetApp NAS server

Using fountain coding for storage

Fountain Coding offer several neat advantages

- **Rateless** - so no need for feedback, timeouts, etc. If a codeword is lost you just have to wait for another to come along.
- **Efficient** - encoding penalty is 3-10% (depending on approach). **ANY** N + ∂ codewords allow you to recover original data.
- **Data can be multicast** - better than simple replication (this allows the data to be read in parallel from many sources at the same time)
- Offers a chance to **load balance** and hence make better use of limited storage and network resources.

Two key drawbacks:

- Potentially computationally expensive. But it is very easy to do in hardware (NetFPGA is a possible solution)
- Storage has to be semi-immutable (e.g. write to erase). Could use a checkpointed git like file system (e.g. Irminsule)

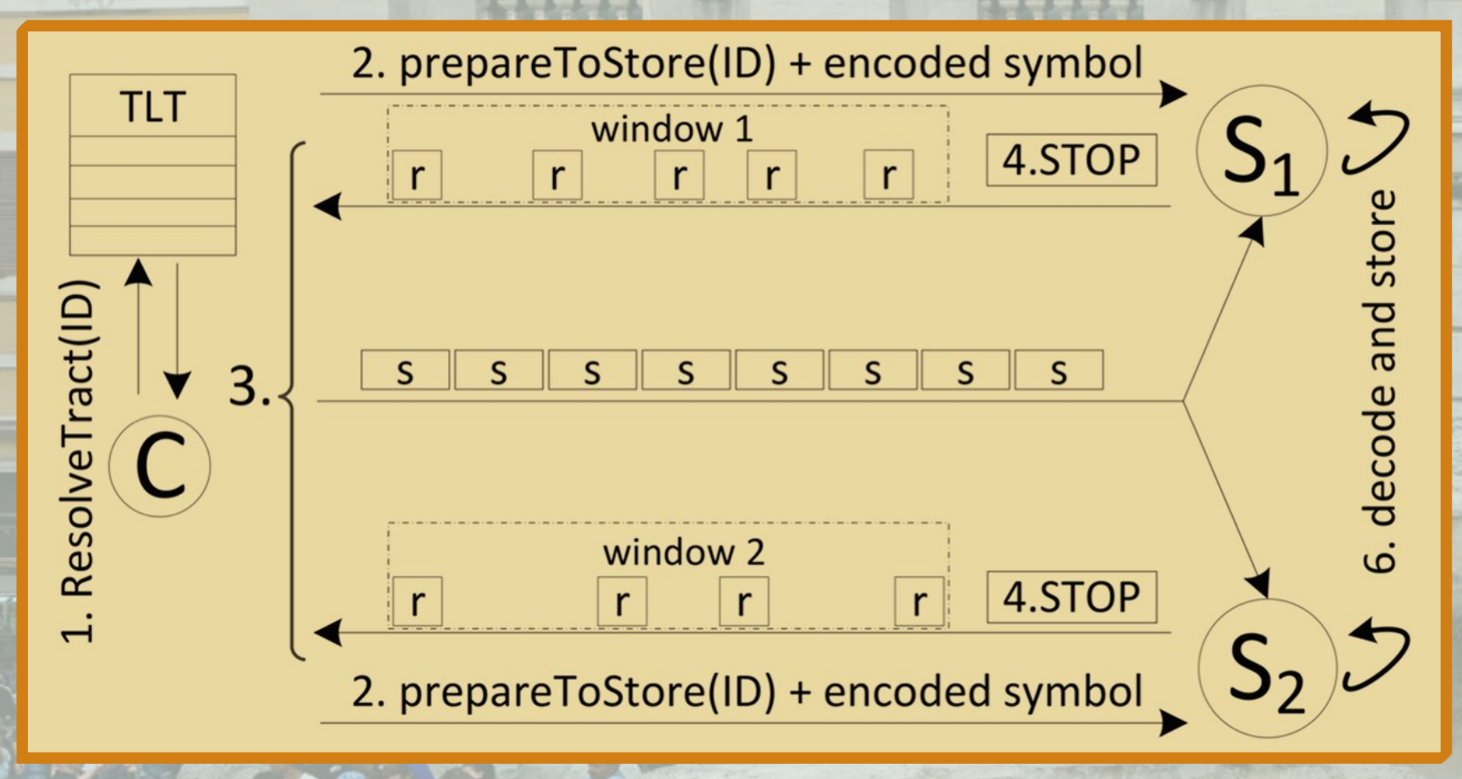
Key mechanisms

Trevi relies on a few key mechanisms

- **Tract Locator Table**. This is based on Flat Datacenter Storage. Table links data with the set of discs and an associated multicast address for them.
- **Multicast**. Trevi uses multicast to create sets of storage nodes. This makes it easier to achieve both **replication** and **multi-sourcing**.
- **Sparse erasure codes**. These allow you to recover data with a predictable small overhead. See the box bottom left for an explanation
- **Receiver-driven flow control**. In a storage system discs can be the bottleneck. So in Trevi receivers explicitly request data at the speed they can process it.

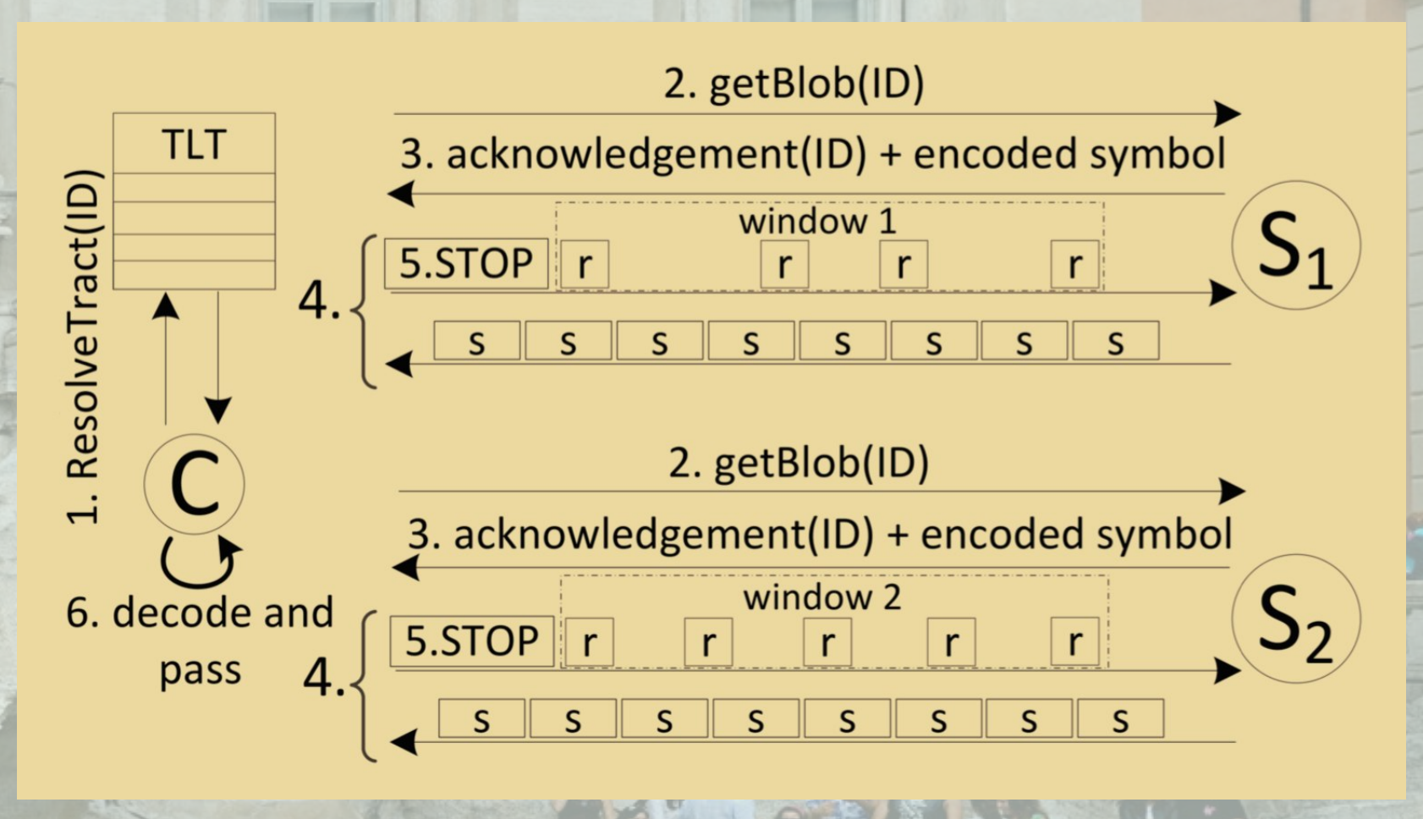
Trevi

Writing data with Trevi



1. Controller decides where blocks are to be stored. Data is converted into code blocks using suitable sparse erasure code.
2. Sender contacts correct storage nodes. Sending a data block improves efficiency.
3. Each receiver sends back a stream of requests for new code blocks. Sender uses these to determine a safe sending rate.
4. Once sufficient codewords received storage node sends stop message. Sender stops sending once all storage nodes have stopped.
5. (not shown) sender can choose to ignore slowest node or otherwise optimise.
6. Storage nodes decode data and store it on their disc array.

Reading data



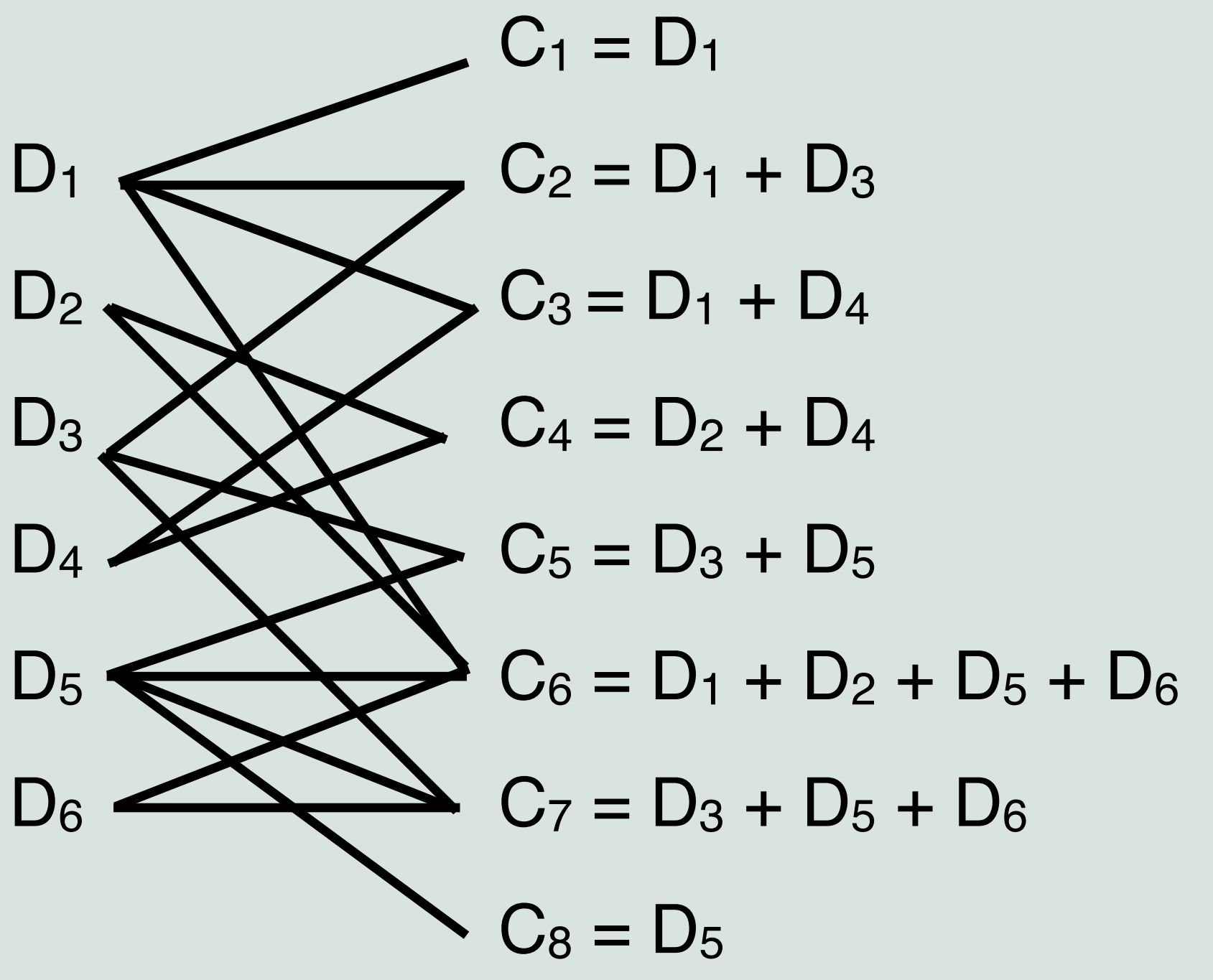
1. Find the set of storage nodes holding the desired data.
2. C sends getBlob request.
3. Each sender recovers correct data and creates set of code words then sends an ack + first code word
4. Stream of code words are sent to C from each node. Random seed at each node prevents the need for any coordination between them
5. Once C has sufficient symbols it stops the senders.
6. Data is decoded and passed up to the correct application at C.

Fountain Coding

Mechanism for multicasting data. Simple idea is data source acts like a drinking fountain:

- To receive a file you simply fill up your cup from a stream of encoded blocks.
- As soon as you have enough blocks your file can be decoded.
- Doesn't matter which blocks you get or the order they arrive in.
- To encode, file is split into chunks. Each chunk is XORed with a selection of others.
- Decoding is simply XORing the coded chunks to recover the data. Some chunks are sent on their own - these act as the keys to unlock the coding.

The clever bit is choosing which chunks to combine to allow efficient decoding.



$C_1 = D_1$
$C_2 = D_1 + D_3$
$C_3 = D_1 + D_4$
$C_4 = D_2 + D_4$
$C_5 = D_3 + D_5$
$C_6 = D_1 + D_2 + D_5 + D_6$
$C_7 = D_3 + D_5 + D_6$
$C_8 = D_5$

Receive codewords C_1, C_2, C_7, C_3, C_4, C_5

Use C_1 to recover $\mathbf{D_1}$
Use C_2 and D_1 to recover $\mathbf{D_3}$
Then wait for C_3
Use C_3 and D_1 to recover $\mathbf{D_4}$
Use C_4 and D_4 to recover $\mathbf{D_2}$
Use C_5 and D_3 to recover $\mathbf{D_5}$
Use C_7 and D_3 to recover ($D_5 + D_6$)
Use ($D_5 + D_6$) and D_5 to recover $\mathbf{D_6}$

Binary and decimal - the basics

- Binary and decimal are both positional number systems.
- Normally we're in base 10 (decimal) so the first position in the number is units, then 10s, then 10^2 (hundreds), then 10^3 (thousands)
- In binary we're dealing in base 2. So first position is units, then 2s then 2^2 (4s) then 2^3 (8s).
- So to get 12 (10 + 2 in base 10) you need 8 + 4 which is 1100 in binary. And to get 14 you need 8 + 4 + 2 which is 1110

XOR (exclusive OR)

- Combine binary numbers. Anywhere with a single 1 becomes a 1 in result, anywhere else is 0.
- This is the same as addition but without any carry.

So 8 + 6 = 14 gives:

$1000 \oplus 0110 = 1110$

while 11 + 6 = 17 gives:

$1011 \oplus 0110 = 1101$ (13)

(so in this example you ignore the carry from the 4 column)

UNIVERSITY OF CAMBRIDGE

Research Systems Group
srg@cambridge:~$

Computer Laboratory
William Gates Building
15, JJ Thomson Avenue
Cambridge, CB3 0FD

EPSRC
Pioneering research and skills

INTERNET
INTelligent Energy awaRe NETworks