PROBLEM STATEMENT

Design a code deployment system for a company that wishes to deploy any web/app/other software services it provides to 100000+ machines spread out across 5-10 regions across the globe in 30 mins tops. (10GB deployed every 30 mins)
Also, you don’t need to worry about the testing code aspect and just design for the building and deploying code aspect

CLARFYING QUES TO ASK

Question 1
Q: What exactly do we mean by a code-deployment system? Are we talking about building, testing, and shipping code?

A: We want to design a system that takes code, builds it into a binary (an opaque blob of data—the compiled code), and deploys the result globally in an efficient and scalable way. We don't need to worry about testing code; let's assume that's already covered.

Question 2
Q: What part of the software-development lifecycle, so to speak, are we designing this for? Is this process of building and deploying code happening when code is being submitted for code review, when code is being merged into a codebase, or when code is being shipped?

A: Once code is merged into the trunk or master branch of a central code repository, engineers should be able to trigger a build and deploy that build (through a UI, which we're not designing). At that point, the code has already been reviewed and is ready to ship. So to clarify, we're not designing the system that handles code being submitted for review or being merged into a master branch—just the system that takes merged code, builds it, and deploys it.

Question 3
Q: Are we essentially trying to ship code to production by sending it to, presumably, all of our application servers around the world?

A: Yes, exactly.

Question 4
Q: How many machines are we deploying to? Are they located all over the world?

A: We want this system to scale massively to hundreds of thousands of machines spread across 5-10 regions throughout the world.

Question 5
Q: This sounds like an internal system. Is there any sense of urgency in deploying this code? Can we afford failures in the deployment process? How fast do we want a single deployment to take?

A: This is an internal system, but we'll want to have decent availability, because many outages are resolved by rolling forward or rolling back buggy code, so this part of the infrastructure may be necessary to avoid certain terrible situations. In terms of failure tolerance, any build should eventually reach a SUCCESS or FAILURE state. Once a binary has been successfully built, it should be shippable to all machines globally within 30 minutes.

Question 6

Q: So it sounds like we want our system to be available, but not necessarily highly available, we want a clear end-state for builds, and we want the entire process of building and deploying code to take roughly 30 minutes. Is that correct?

A: Yes, that's correct.

Question 7
Q: How often will we be building and deploying code, how long does it take to build code, and how big can the binaries that we'll be deploying get?

A: Engineering teams deploy hundreds of services or web applications, thousands of times per day; building code can take up to 15 minutes; and the final binaries can reach sizes of up to 10 GB. The fact that we might be dealing with hundreds of different applications shouldn't matter though; you're just designing the build pipeline and deployment system, which are agnostic to the types of applications that are getting deployed.

Question 8
Q: When building code, how do we have access to the actual code? Is there some sort of reference that we can use to grab code to build?

A: Yes; you can assume that you'll be building code from commits that have been merged into a master branch. These commits have SHA identifiers (effectively arbitrary strings) that you can use to download the code that needs to be built.

ACID transactions will make it safe for potentially hundreds of workers to grab jobs off the queue without unintentionally running the same job twice (we'll avoid race conditions). Our actual transaction will look like this:

```
  BEGIN TRANSACTION;
  SELECT * FROM jobs_table WHERE status = 'QUEUED' ORDER BY created_at ASC
LIMIT 1;
  // if there's none, we ROLLBACK;
  UPDATE jobs_table SET status = 'RUNNING' WHERE id = id from previous
query;
  COMMIT;
```

The transaction that this auxiliary service will perform will look something like this:

```
  UPDATE jobs_table SET status = 'QUEUED' WHERE
    status = 'RUNNING' AND
    last_heartbeat < NOW() - 10 minutes;
```

Date:
M T W T F S S

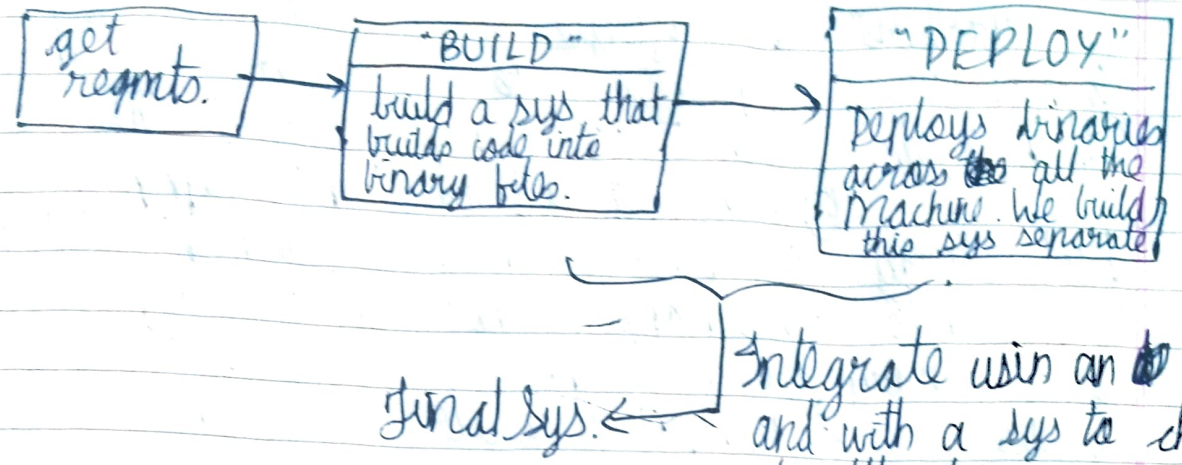
DESIGN - CODE DEPLOYMENT SYSTEM

(A) General Reqmts. for sys

Design	Info	Reason
Sys = code deployin sys	Repeatedly (10^3+ times/day) builds, deplogs code to 10^5+ machines across 5-10 regions around the world	
Buildin code is grabbin snapshots of source code upin SHA identifiers	Implementn details of bldg acn is not somethin we need to take care of	eg: we don't need to worry how we build JS or C++ code. Our sys → only enables repeated bldg of code.
Need to split up sys in 2 parts 1) Bldg code → takes upto 15 mins, res = binary file of upto 10GB 2) Deplogin code → another 10-15 mins to pass built code to target machines		as per reqs we want entire process to happen under 30 mins
Each build will need a clear end state → i) Success or ii) Failure or iii) Runnin	Availability for sys → 2-3 nines	Don't need to optimize availability as of now

Date:
M T W T F S S

(B) Sys Design Plan



(C) PLAN IMPLEMN

[I] THE "BUILD" SYS

Decision	Info	Reason
(1) General Overview		
Build sys = queue of "jobs"	Job → process of bldg code into a binary	To ensure proper chronological order in which code is built & eventually deployed
Each job has a commit identifier (SHA identifier)	SHA → Secure Hasin Algo. converts code into 'encrypted code'	Identifier tells worker what version of code it shud build and the name of the "artifact" (name of resultin binary) that "ll be created.
Assump" → all langs are auto handled automatically		We don't need to des. for type of code being built

Pool of servers known as "workers" to handle all these jobs

Blob storage is chosen to store binary blobs

① Each worker takes jobs off the queue repeatedly (using FIFO princip)
② Then, it'll build the binaries (how this happens → not our prob, rn)
③ Then, it'll write resultin binaries (a.k.a. artifacts) to blob storage (eg:- Google Cloud Storage, Amazon S3)

This way, we ensure that code is built in a chronological order which makes it easier to query it later using its timestamp and thus deployment will also be chronological

Cuz binaries are literally blobs of data so blob storage is apt. Binaries recd by workers are stored in blob storage in

② JOB QUEUE

SQL dB is chosen

Queue will be implemented as an SQL dB

i) We wanna store historical status in a soln. that, unlike in-mem, persists thru server failures.

Each "job" is a row/record in this SQL table

The attrs/cols per "job" are
① id : string : the ID of job, auto generated
② created_at : a timestamp as to when job was passed for buildin
③ last_heartbeat

For proper queryin of each single "job"

④ commit_SHA : str : the commit SHA identifier
⑤ name : str.
⑥ status : str
→ QUEUED
→ RUNNIN
→ SUCCEEDED
→ FAILED

ptr to job's eventual binary in blob storage

idx ↑ ↑ idx

	id	created-at	commit_SHA	name	status	last_heart beat
job 1 ⇒						
job 2 ⇒						
job 3 ⇒						

general overview of queue implemented as SQL table

NOTE:

1) Dequeuing mech. applied by looking at the oldest created_at (timestamp) with a status = QUEUED

2) ∴ We gotta idx table at → created_at, status

Decision	Info	Reason
(3) SQL tables ACID transac^n help ensure no concurrency happens while dequeuein (i.e. no 2 workers tryna grab the same job)	All workers dequeue next job every 5 sec. If we have 100 workers, we are makin 100/5 = 20 queries to dB per sec. which is easily handled by an SQL dB	Strong Consistency ensured by ACID transac^n keeps dB updated (i.e. as soon as 1 worker grabs a job, its status is set to RUNNING so other worker won't grab it

(4) Edge Case → Lost Jobs

last_heartbeat col. for every job in SQL table. It's basically updates on status of job every small duration and is stored separately

Std. time to build code = 15 mins

Worker updates job's last heartbeat (of the job it's building) every 3-4 mins i.e.

Updates help us know abt the 'health of a server' to deal with cases where server dies mid-built or suffers a network partition or some other failure

We use a separate service to continuously monitor the last_heartbeat

If last_heartbeat was modified longer than 2 heartbeats before (i.e. 10 mins), somethin wrong has happened with server (died, isolated etc.)

↓ this separate service

resets status of the job from RUNNING to QUEUED

(5) SCALE ESTIMAN

≈100 workers reqd to build code.

No. of builds per day = $5000 - 10^4$

No. of builds by each worker per:
a) hr = 4
∴ day = $4 \times 24 = 96$

∴ No. of workers = [No. of builds / (No. of builds by 1 worker)]
= 10000/96
≈ 100

If builds aren't uniformly spread out across the day i.e. eg:- 8000 builds in day, 2000 at night, we gotta scale workers hori. ally or vertically to handle load.

P2P = peer-to-peer

(6). STORAGE		
Binaries read from workers for each job stored to blob storage	Job status col. in SQL table updated to SUCCEEDED	Worker is released and will now look for the oldest status=QUEUED waala job
Regional clusters of storage i.e each region will have its own blob store	Regional blob stores replicate binaries stored in main blob store asynchronously	Faster access to data for each region

II DEPLOYMENT SYS

Decision	Info	Reason
(1) GENERAL OVERVIEW		
3 things read.		
Peer-to-Peer Network in each region	This P2P network will easily deploy binary stored in this region's blob store very fast	P2P networks help distri. data from 1 storage to several machines real fkin fast
Auxillary service 1	Tells us when a binary has been replicated	Once this service tells 'binary has been replicated!', P2P network starts its work.
Auxillary Service 2	Serves as a 'source of truth' for what binary should be currly run on all machines	

(2) REPLICAN STATUS SERVICE (auxiliary service 1)

Global service to check all regional blob stores a.k.a. buckets

Once a binary has been replicated across all buckets, this service updates a separate SQL DB where

① row = name of regional bucket

② col. 1 = name of built binary

③ col. 2 = replicaⁿ. status ⇒ complete, failed, pending

Only once a binary is replicated across all buckets i.e. replicaⁿ status col. in SQL DB is completed for all rows, only then is it officially deployable.

(3) P2P NETWORK

P2P network relaⁿship implemented : regional clusters i.e. cluster of machines in that region we wanna deploy to

10GB file in regional by

regional P2P

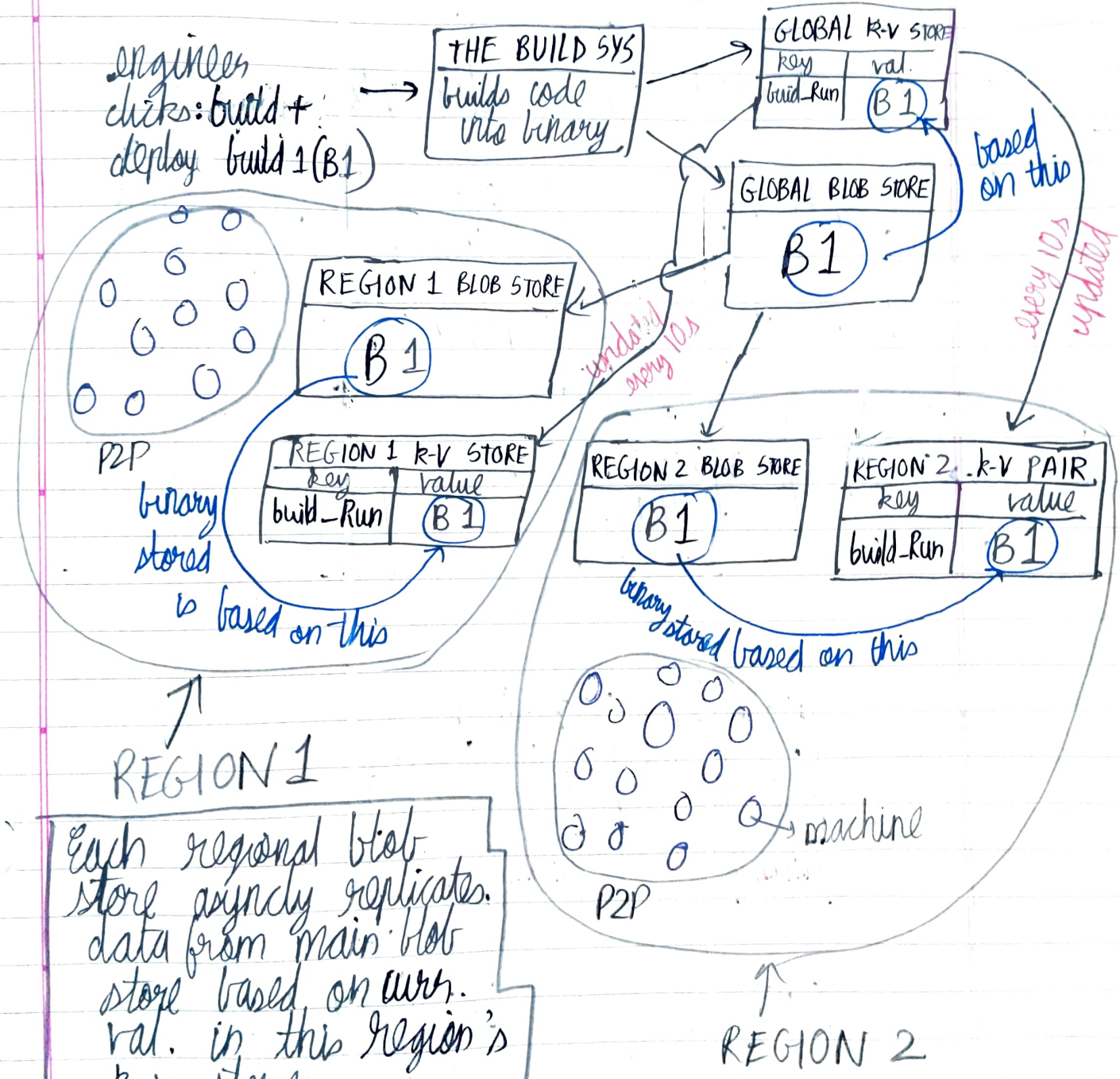
distributes entire file ∞ in "regional cluster" very fast

Cuz P2P networks are fd motherfkin fast.

(4) AUXILLARY SERVICE 2 → TRIGGER

Reason ⇒ We basically implement this to ensure supportin deployment of multip builds.

Implementa" ⇒ K-V store used for every region as well as our main (a.k.a. "global") k-V store holds config" abt what build should be curr. ly deployed in this region's cluster.



Each regional blob store asyncly replicates data from main blob store based on curr. val. in this region's k-V store.

Handling multiple builds

(1) Some other engineer, whilst this build + deploy is happening, clicks → build + deploy build 2. (B2)

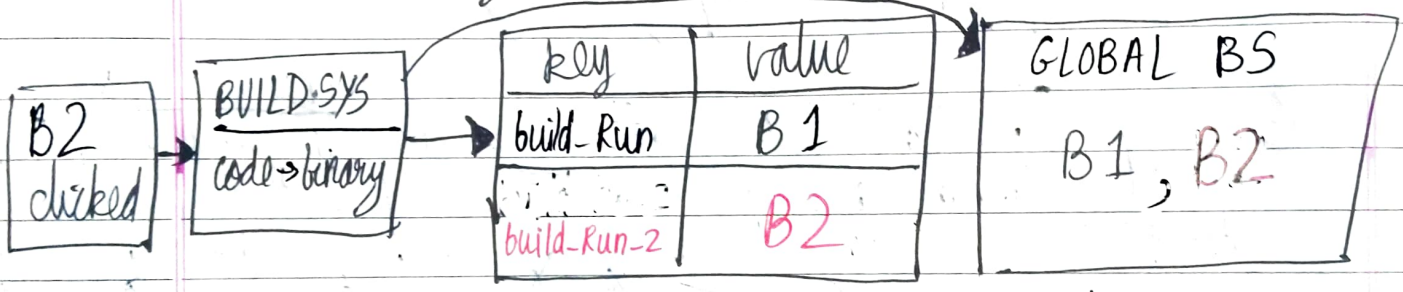
(2) Another k-v pair added in global blob store

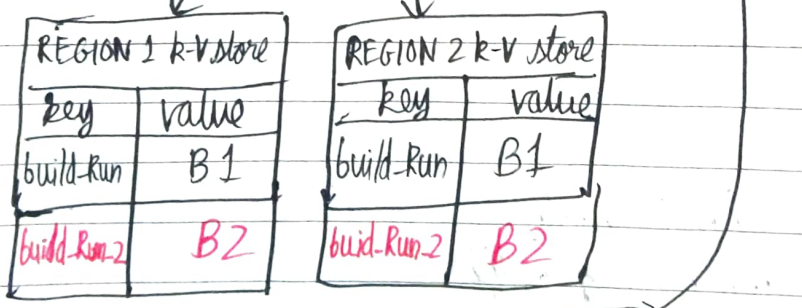
build_Run_2	B2
↑ key	val

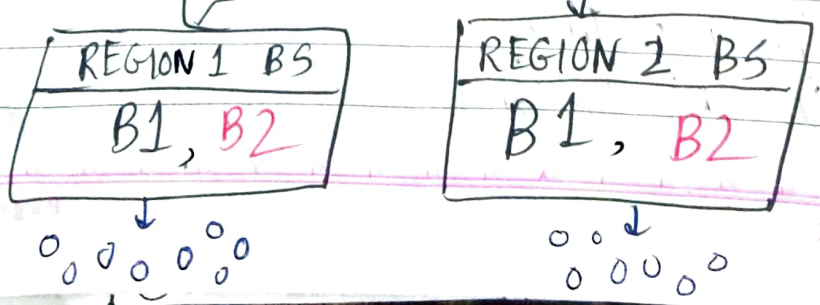
(3) B2 loaded concurrently into global dB

(4) Every 10s, regional k-v stores query global k-v store for any new k-v pair

Now, they also replicate this k-v pair



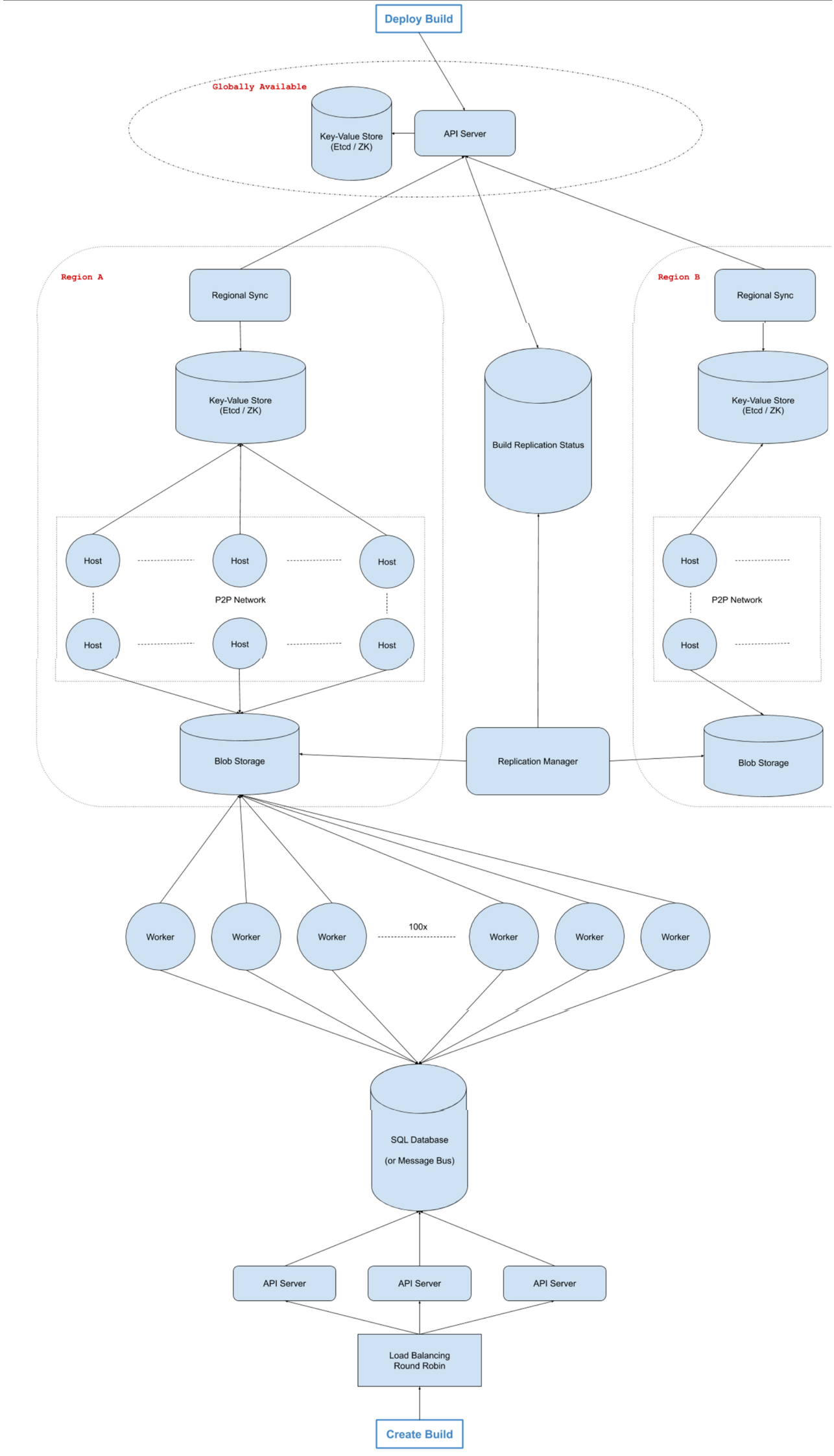




Now, they tell their region's blob store to replicate the binary (B2) as per the k-v pair just added

(5) Each regional BS now replicates B2 binary simult. whilst its say, distributin B1 binary.

(6) regional BS deploys new binary B2 to machines via P2P network


Deploy Build
Globally Available
Key-Value Store
(Etcd / ZK)
API Server
Region A
Regional Sync
Key-Value Store
(Etcd / ZK)
Build Replication Status
Region B
Regional Sync
Key-Value Store
(Etcd / ZK)
Host
Host
Host
P2P Network
Host
Host
Host
Host
P2P Network
Host
Blob Storage
Replication Manager
Blob Storage
Worker
Worker
Worker
100x
Worker
Worker
Worker
SQL Database
(or Message Bus)
API Server
API Server
API Server
Load Balancing
Round Robin
Create Build