Tema 3. Diseño de Software Crítico

Manuel Díaz mdiaz@uma.es

Dpto. Lenguajes y Ciencias de la Computación
Universidad de Málaga

Diseño de Software Crítico I

- Patrones para gestión de errores
    - Detección de anomalías
    - Manejo de errores: Rejuvenation y Bloques de Recuperación
- Replicación y diversificación
- Balances arquitectónicos: disponibilidad/fiabilidad, seguridad/prestaciones,…

Why Detect Errors?

- Given the fault->error->failure chain considered in safety critical systems, it can be useful in a running system to detect the error caused by the (undetected) fault and take appropriate action before it becomes a failure
- Once the system is running, it is too late to detect faults, but errors often leave observable traces
- By detecting the error, it may be possible to avoid a failure completely, but even if it is not, it may be possible to log details of the problem and inform the larger system into which the failing component is embedded before the failure occurs

Error Detection and Standards

- Some techniques for detecting an error, as listed in Table 4 of ISO 26262-6, are straightforward and likely to be applied to any software development (e.g functions performing range checks on their input parameters)
- Others, such as “plausibility" checking, are more subtle and are considered in this chapter
- Table A.2 of IEC 61508-3 recommends the incorporation of fault detection into a software design (from the description, the techniques are more focused on error detection than on fault detection)

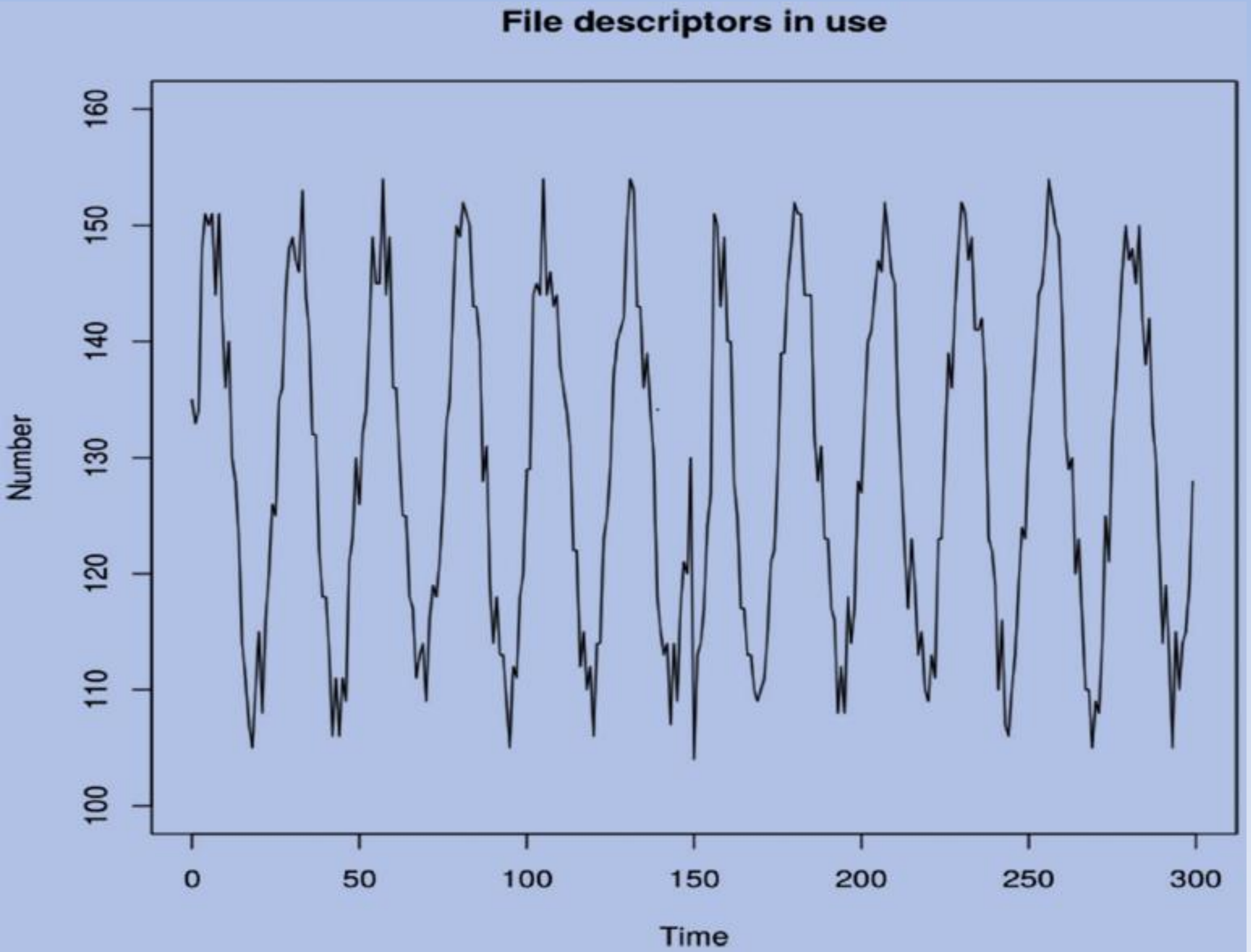
Anomaly Detection

- Anomaly detection is the application of an algorithm to a series of events to decide whether one or more of the events is unexpected in the light of the history of the events up to that point

- Humans are very good at this: One glance at a run-chart is enough for a human to know that, even with a noisy signal, something strange happens

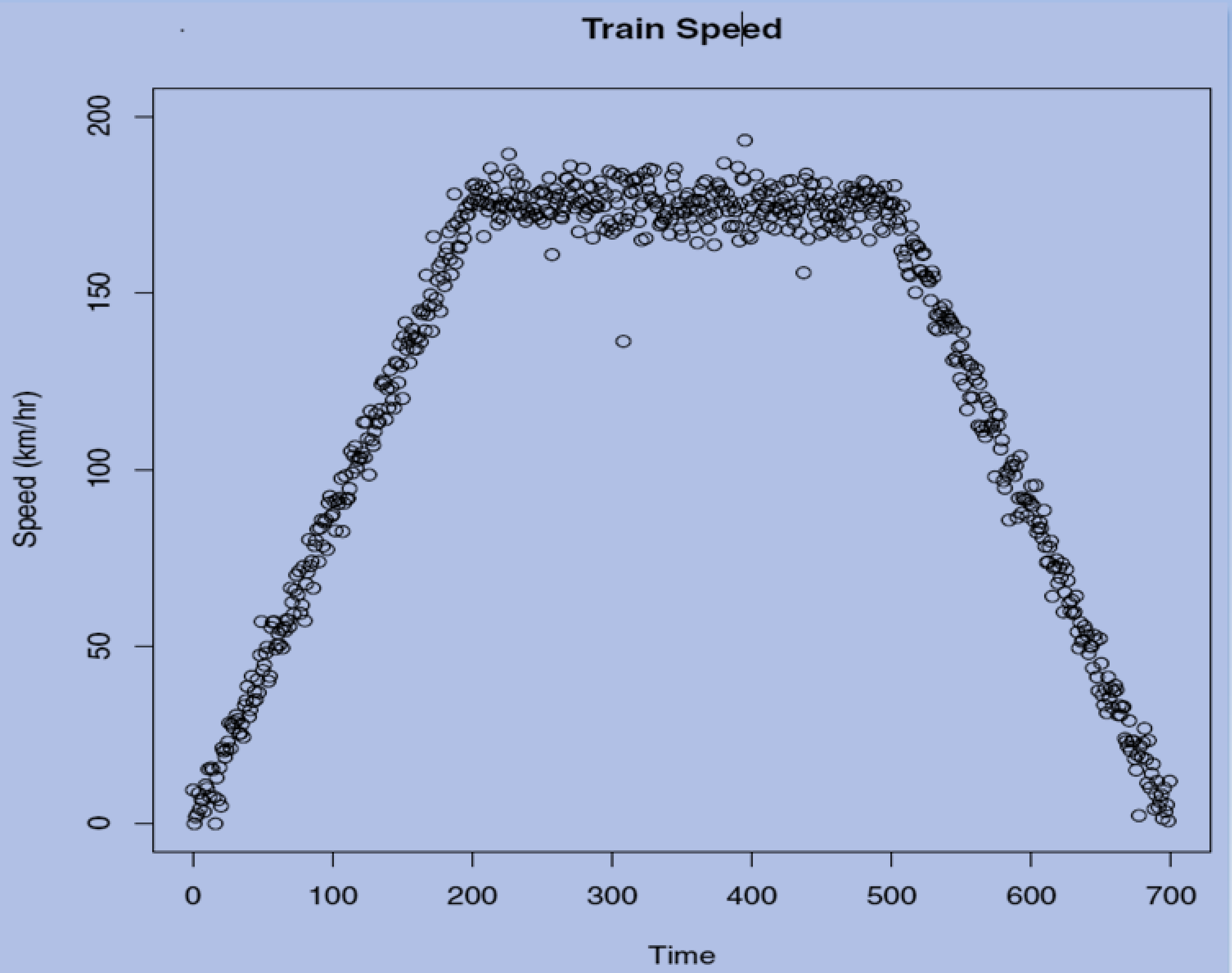
- We will consider different types of algorithms for detecting such anomalies and deciding how anomalous they are

Anomaly Detection. Examples



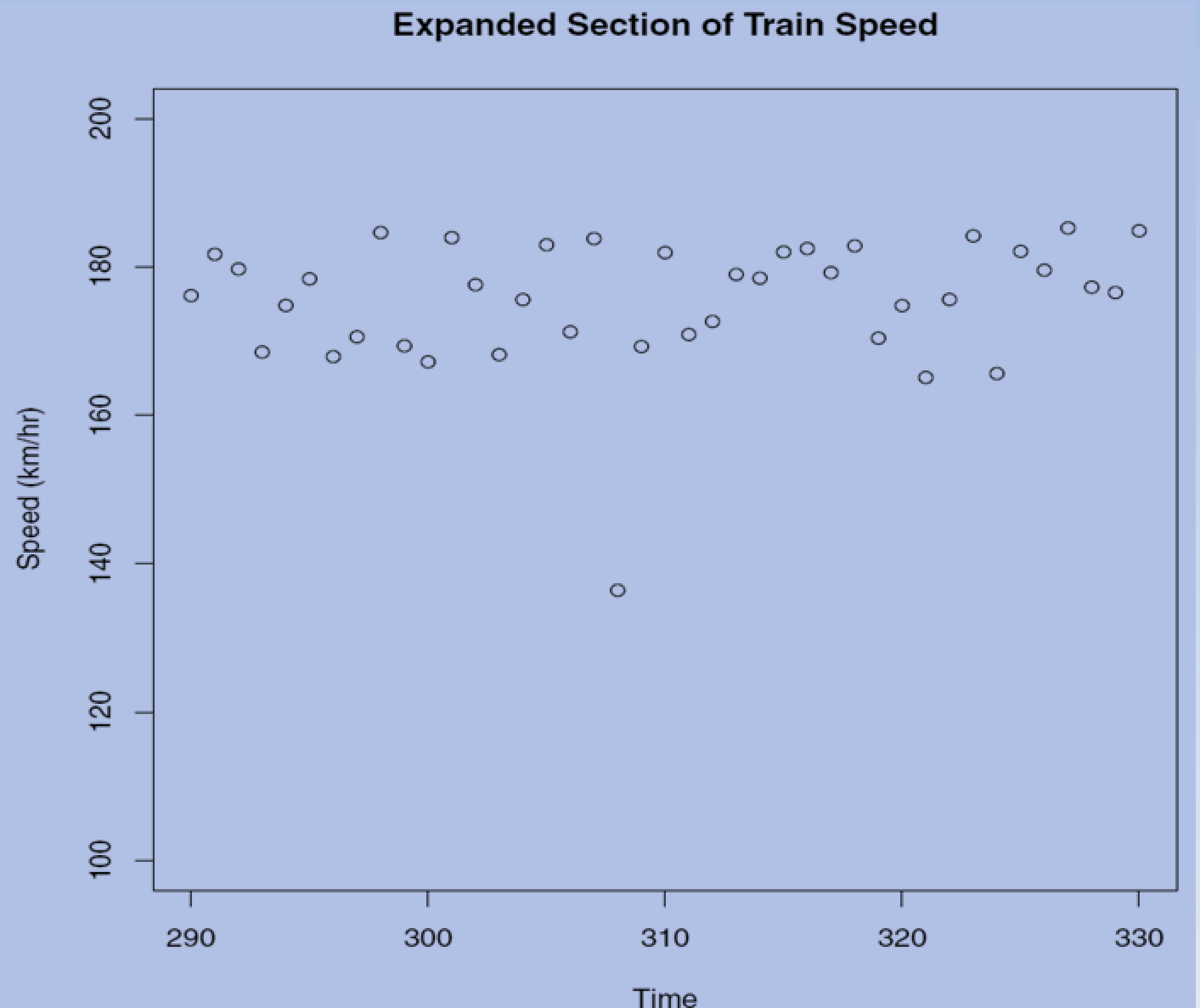
- The values in the figure represent the exact number of an operating system resource type in use over time
- We have assumed that the resource is an open file descriptor, but it could be an amount of memory, a number of locked mutexes, or any other resource whose usage can be measured precisely
- Clearly something odd happens around time t = 150

Anomaly Detection. Examples



- This figure presents a different type of anomaly. The values are assumed to be the speed measurements from a forward-facing Doppler radar on a train
- The train accelerates to around 170 km/h, cruises at that speed for a while and then comes to a stop
- In contrast with the other example, where the values were exact counts, there is some noise on the readings caused by the nature of the radar and there are some values that, to the eye, immediately appear to be anomalous: particularly those at t = 308 and t = 437.

Anomaly Detection. Examples



- We can easily pick out the anomalies in the examples because we can see not only what occurred before them, but also what happened afterwards
- The questions we need to address for the running system are

"while receiving the values from the radar at time t =308, without the benefit of seeing into the future, is the reading of 136.5 km/hr that we have just received anomalous? If so, how anomalous?"

Anomaly Detection

- Two types of anomalies can be identified:
  - point anomalies, such as those which occur in the train speed example at time 308, where a few unexpected points occur
  - contextual anomalies, such as that in the file descriptor example around time 150, where the points by themselves are not anomalous, but the pattern is
- We review some techniques for detecting anomalies at run-time
- Much of the research being carried out on anomaly detection is aimed toward the detection of malicious intrusions into networks by attackers and the examination of financial databases, looking for patterns of fraud

Supervised Anomaly Detection

- Supervised anomaly detection algorithms are trained on a labeled dataset, where the anomalous instances are known.
- The algorithm learns from this labeled data to identify anomalous instances in new, unseen data.
- Some of the most commonly used supervised anomaly detection algorithms include:
  - Support Vector Machines (SVMs):
  - Decision Trees
  - Logistic Regression

Supervised Anomaly Detection

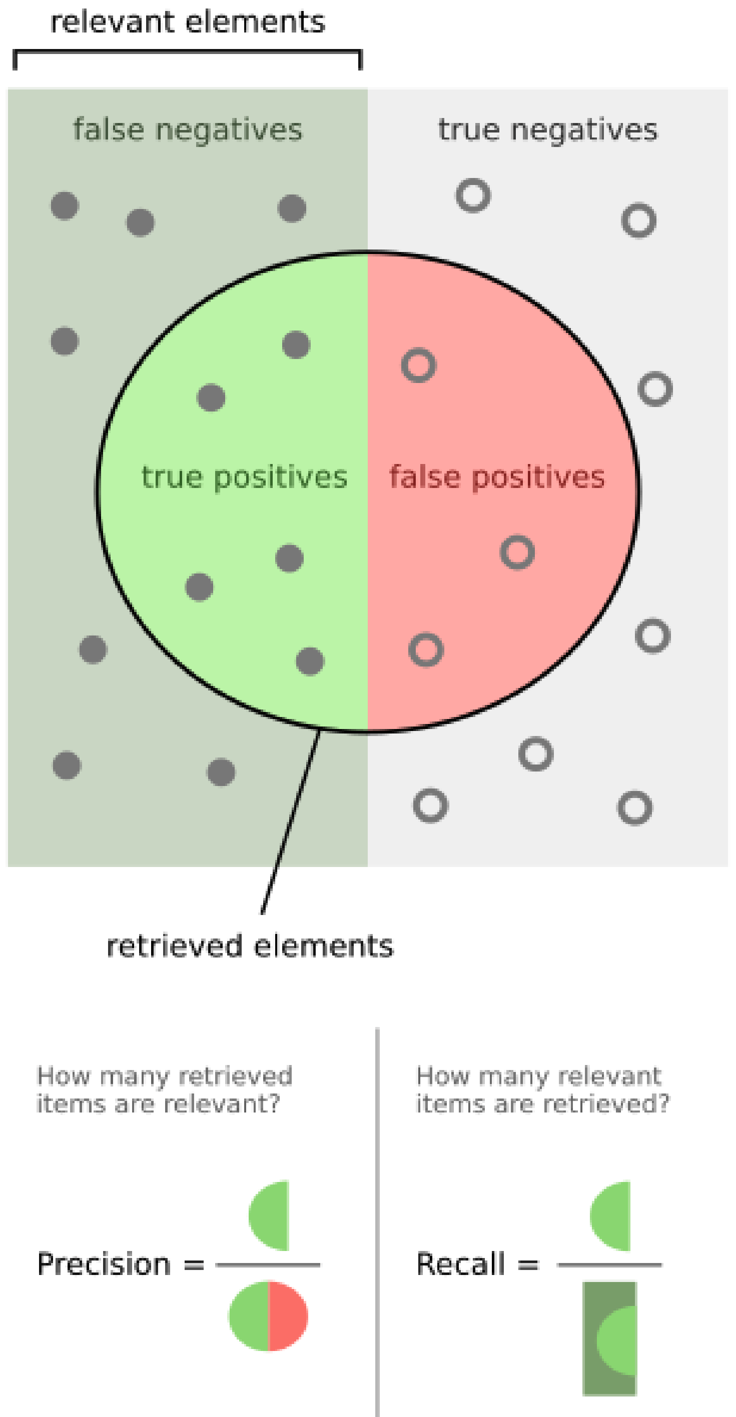
- Decision Trees:
  - Decision trees are a type of machine learning algorithm that can be used for both classification and regression problems. In anomaly detection, a decision tree can be trained to identify anomalous instances in the data.
- Logistic Regression:
  - Logistic regression is a statistical method that is used to model the relationship between a dependent variable and one or more independent variables. In anomaly detection, logistic regression can be used to determine the probability that a given instance is anomalous.
- Support Vector Machines:
  - SVMs are commonly used for binary classification problems, where the goal is to separate two classes in a data set.
  - The SVM algorithm can be adapted for use in anomaly detection by treating the normal instances as one class and the anomalies as another.

Unsupervised Anomaly Detection

- Unsupervised anomaly detection algorithms do not use labeled data to identify anomalies.
- Instead, they rely on the intrinsic properties of the data itself to identify unusual instances. Some of the most commonly used unsupervised anomaly detection algorithms include:
  - Clustering
  - Density-based methods
  - Statistical methods
- Nos centraremos en este tipo de métodos y en especial los basados en aprendizaje

Unsupervised Anomaly Detection

- **Clustering:**
  - Clustering algorithms group instances in a data set based on similarity. In anomaly detection, the idea is to identify instances that do not belong to any of the clusters as anomalies. K-means is a commonly used clustering algorithm.
- **Density-based methods:**
  - Density-based methods rely on the assumption that normal instances in the data form a dense cluster. Anomalies, on the other hand, are instances that are far away from the dense cluster. DBSCAN is a popular density-based method.
- **Statistical methods:**
  - Statistical methods for anomaly detection are based on the assumption that normal instances in the data follow a certain probability distribution. Instances that deviate significantly from this distribution are considered anomalies. Z-score and Mahalanobis distance are two commonly used statistical methods for anomaly detection.



Evaluation of Anomaly Detection Techniques

- Precision and Recall: metrics used to evaluate the performance of anomaly detection algorithms
  - **Precision** is the fraction of instances that are correctly identified as anomalies out of all instances that have been identified as anomalies.
  - **Recall**, on the other hand, is the fraction of instances that are correctly identified as anomalies out of all instances that are actually anomalies.
- Precision and recall are related metrics and a trade-off exists between the two.
  - **High precision** means that the algorithm is able to identify fewer false positives (instances that are not actually anomalies but are identified as such)
  - **High recall** means that the algorithm is able to identify more of the actual anomalies.

Evaluation Anomaly Detection Techniques

- Accuracy
  - Accuracy is a commonly used metric to evaluate the performance of machine learning algorithms.
  - In the context of anomaly detection, accuracy can be defined as the fraction of instances that have been correctly identified as normal or anomalous.
  - While accuracy can be a useful metric, it can be misleading when applied to anomaly detection algorithms. This is because it does not take into account the imbalance between the number of normal and anomalous instances in the data.

Evaluation Anomaly Detection Techniques

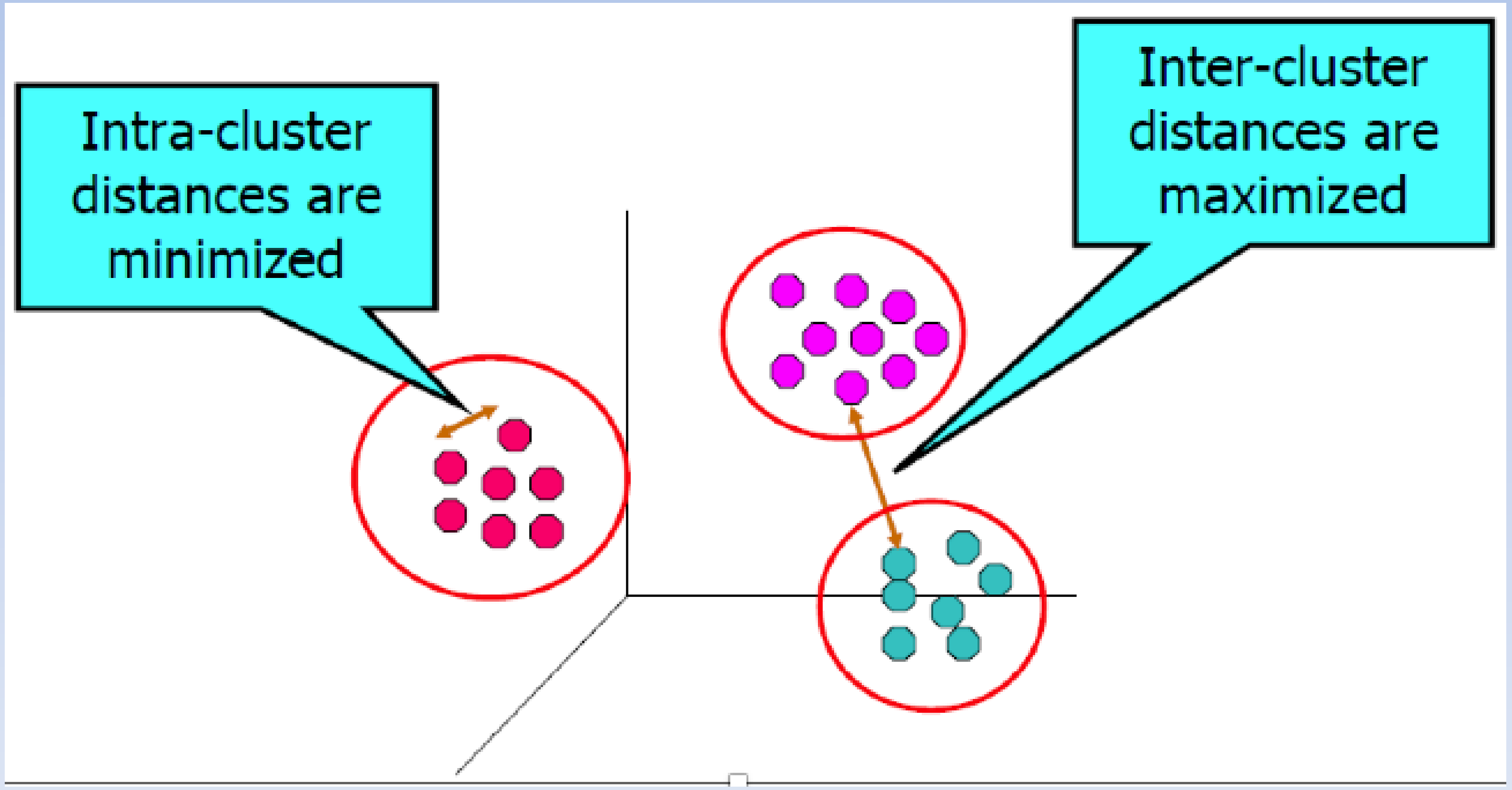
- F1 Score
  - The F1 score is a metric that is used to balance precision and recall. It is the harmonic mean of precision and recall and is calculated as

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall}$$

  - The F1 score provides a single metric that summarizes the balance between precision and recall and is a useful metric for evaluating the performance of anomaly detection algorithms

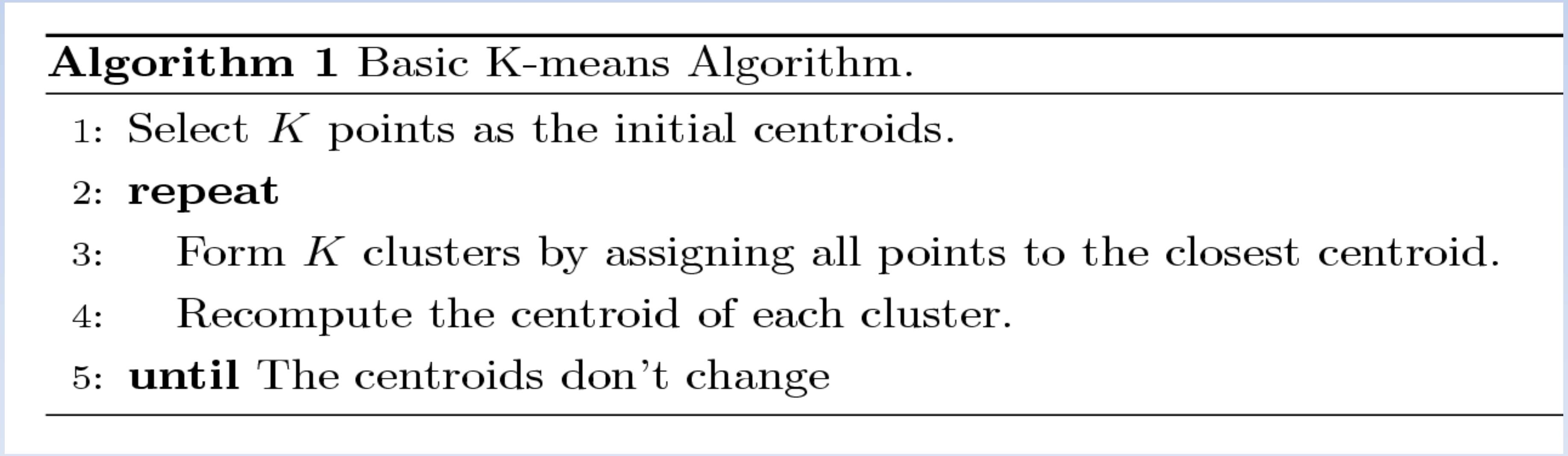
Clustering

- Finding groups of objects such that the objects in a group will be similar to one another and different from the objects of the other


Intra-cluster distances are minimized
Inter-cluster distances are maximized

K-means Clustering

- Each cluster is associated with a centroid (center point)
- Each point is assigned to the cluster with the closest centroid
- Number of clusters must be specified
- The basic algorithm is very simple

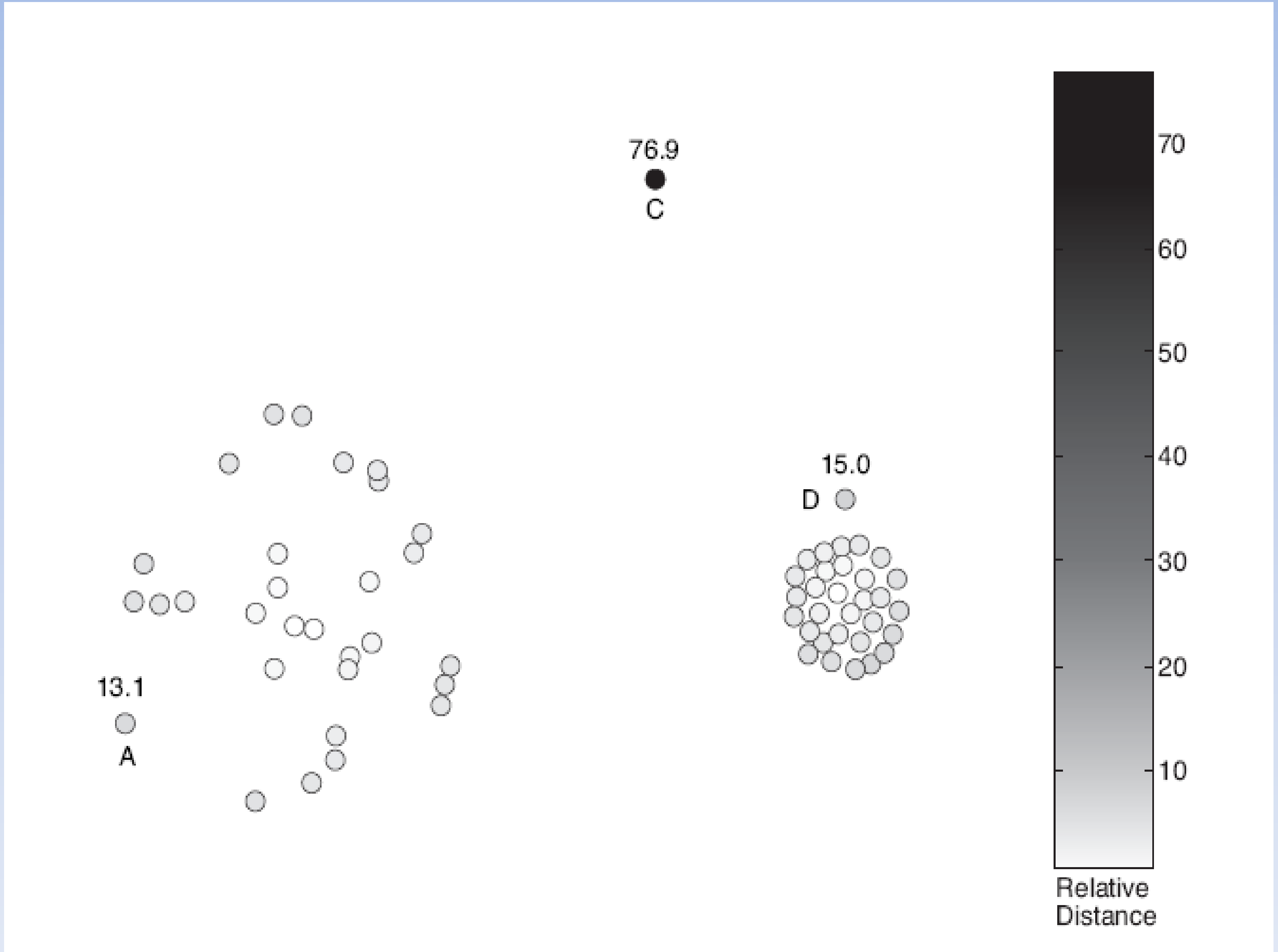


Algorithm 1 Basic K-means Algorithm.

```
1: Select $K$ points as the initial centroids.
2: repeat
3:     Form $K$ clusters by assigning all points to the closest centroid.
4:     Recompute the centroid of each cluster.
5: until The centroids don't change
```

Clustering fo Anomaly Detection

- Outlier score defined as relative distance
- It is the ratio of the distance of point from the closest centroid to the median distance of all points in the cluster from the centroid.



Clustering for Anomaly Detection

Algorithm 2 Basic clustering-based algorithm of anomaly detection

1. Find position of K centroids (Algorithm 1)
2. Find median distance for each centroid
3. Calculate the score of each point that is

(dist of point to nearest centroid)/(median dist. for given nearist centroid)

1. Order the scores to define outliers.

On-line Clustering fo Anomaly Detection

- To the extent that clustering takes place in the brain, it happens in an on-line manner: each data point comes in, is processed, and then goes away never to return

- To formalize this, imagine an endless stream of data points x_1, x_2, . . . and a k-clustering algorithm that works according to the following paradigm:

```
repeat forever:
get a new data point x
update the current set of k centers
```

On-line Clustering for Anomaly Detection

- This algorithm cannot store all the data it sees, because the process goes on ad infinitum.

- More precisely, we will allow it space proportional to k. And at any given moment in time, we want the algorithm's k-clustering to be close to the optimal clustering of all the data seen so far

- There are many different approaches, but they are not very used in this context

Isolation Tree

- They are known to work well for high dimensional data.
- An Isolation forest is a ensemble of “Isolation Trees”
- An Isolation tree is a binary tree that stores data by dividing it into boxes (called nodes)

Algorithm Isolation Tree

1.Select a feature at random from data. Let us call the random feature f.

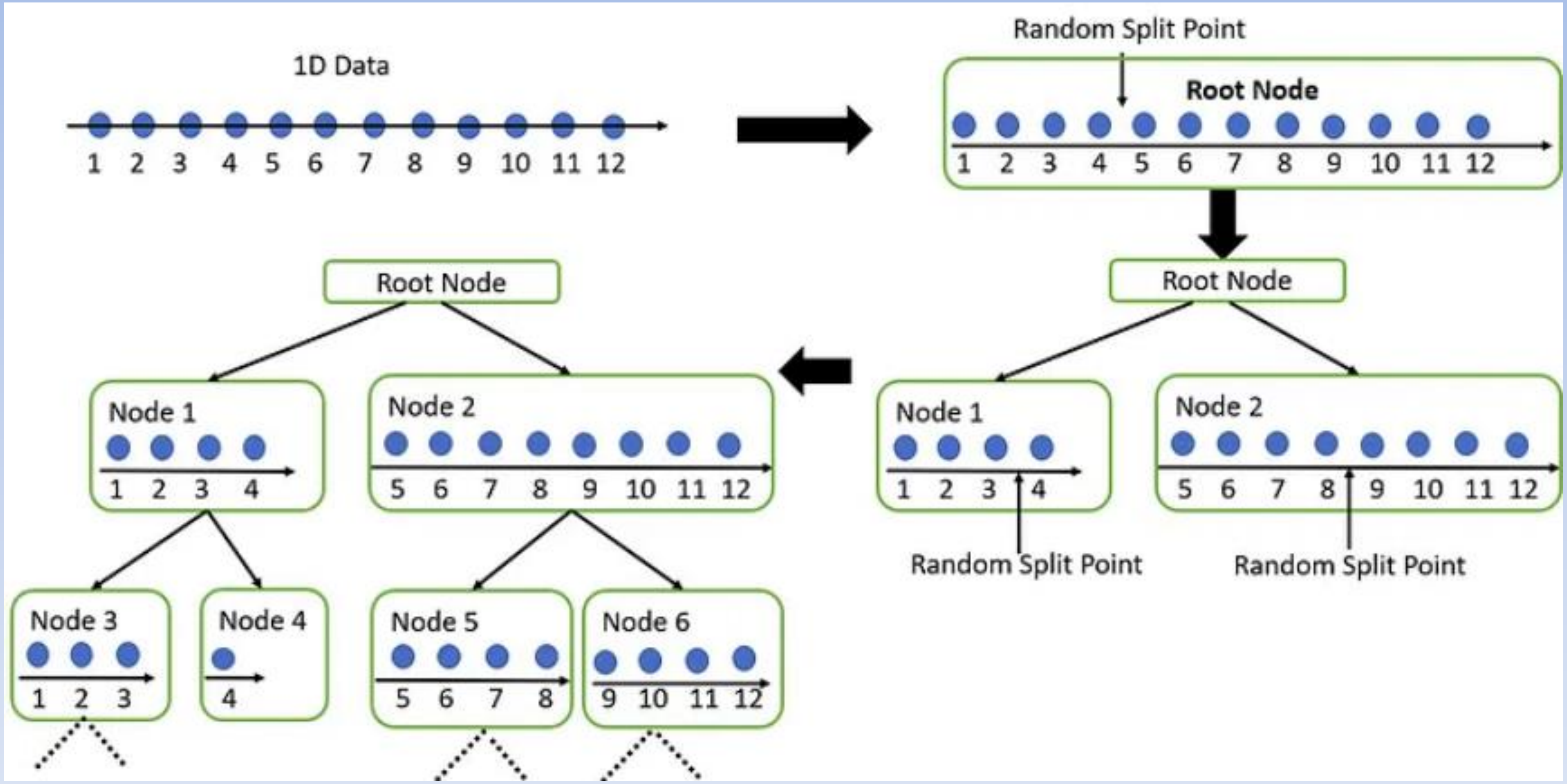
2.Select a random value from the feature f. We will use this random value as a threshold. Let us call it t.

3.Datapoints where f <t are stored in Node1
wheref ≥t go in Node2.

1. Repeat Steps 1–3 for Node 1 and Node 2.

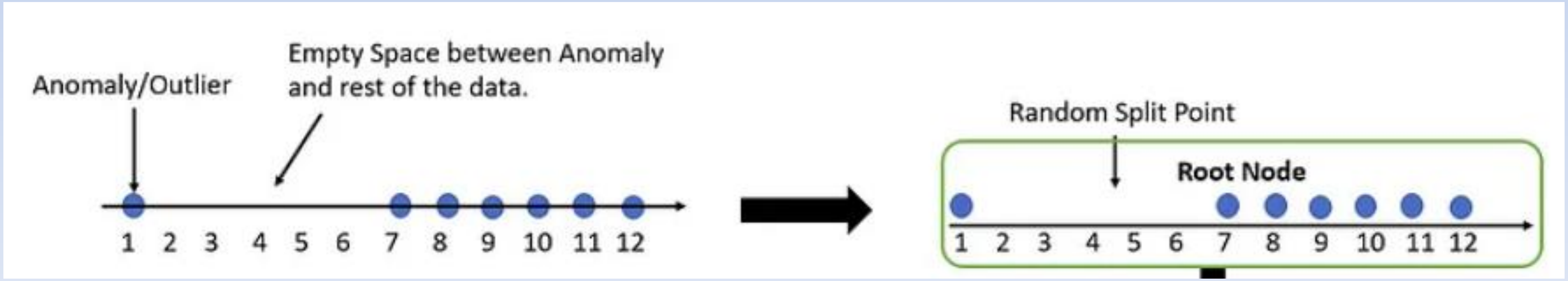
2.Terminate either when the tree is fully grown or a termination criterion is met.

Isolation Tree – No anomaly

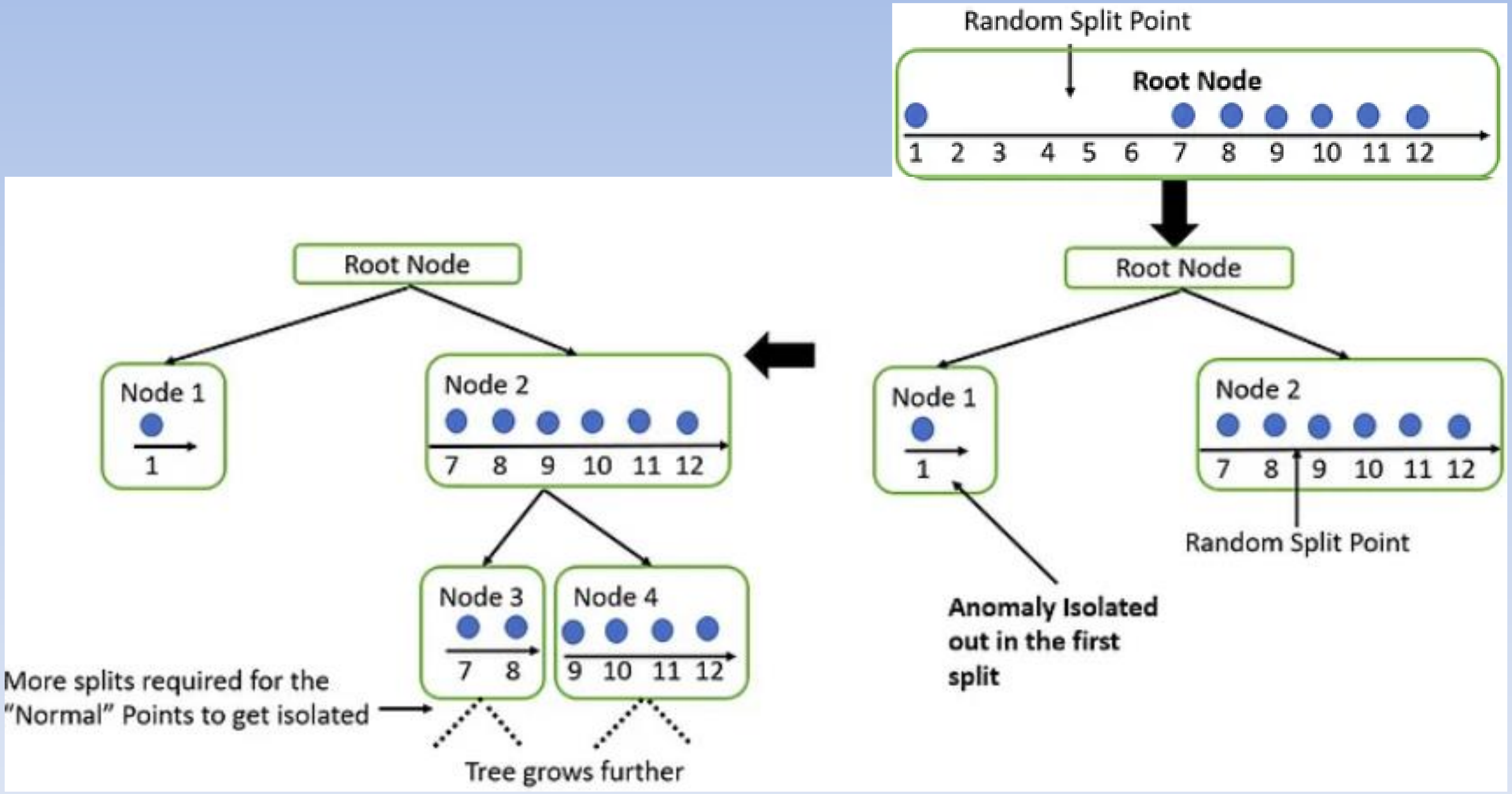


Isolation Tree

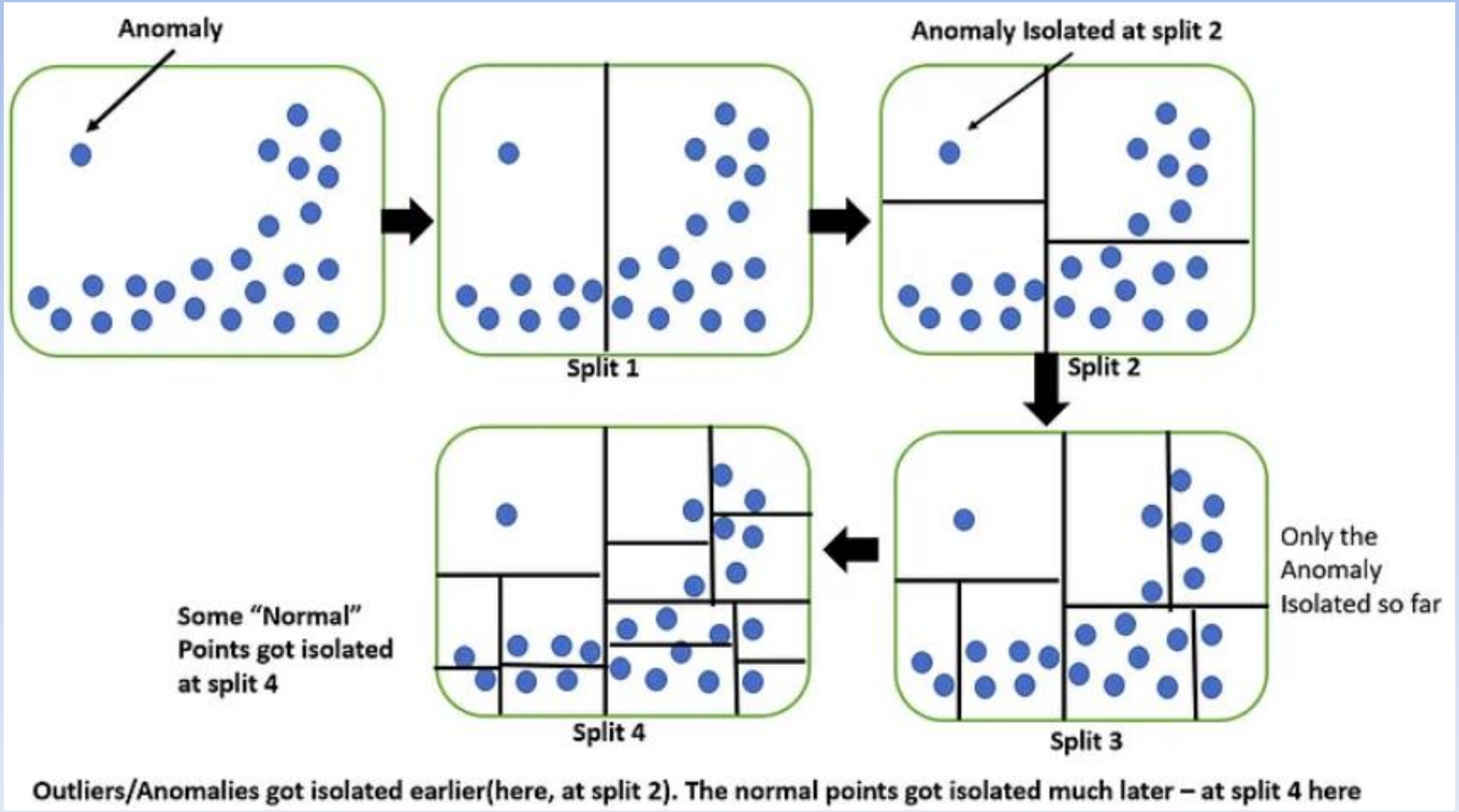
- Now, assume the univariate data above has an anomaly. In that case, the anomalous point will be far away from the other data points
- Isolation forests are able to isolate out anomalies very early on in the splitting process because the Random Threshold used for splitting has a large probability of lying in the empty space between the outlier and the data if the empty space is large enough.
- As a result, anomalies have shorter path lengths. After all, the split point(the threshold is chosen at random. So, the larger the empty space, the more likely it is for a randomly chosen split point to lie in that empty region.



Isolation Tree



Isolation Tree - Multivariate


Anomaly
Anomaly Isolated at split 2
Split 1
Split 2
Only the Anomaly Isolated so far
Split 3
Split 4
Some "Normal" Points got isolated at split 4

Outliers/Anomalies got isolated earlier(here, at split 2). The normal points got isolated much later – at split 4 here

Isolation Forest

- We can extend the idea of an Isolation tree to an isolation forest which is an ensemble of multiple Isolation trees

Algorithm Isolation Forest

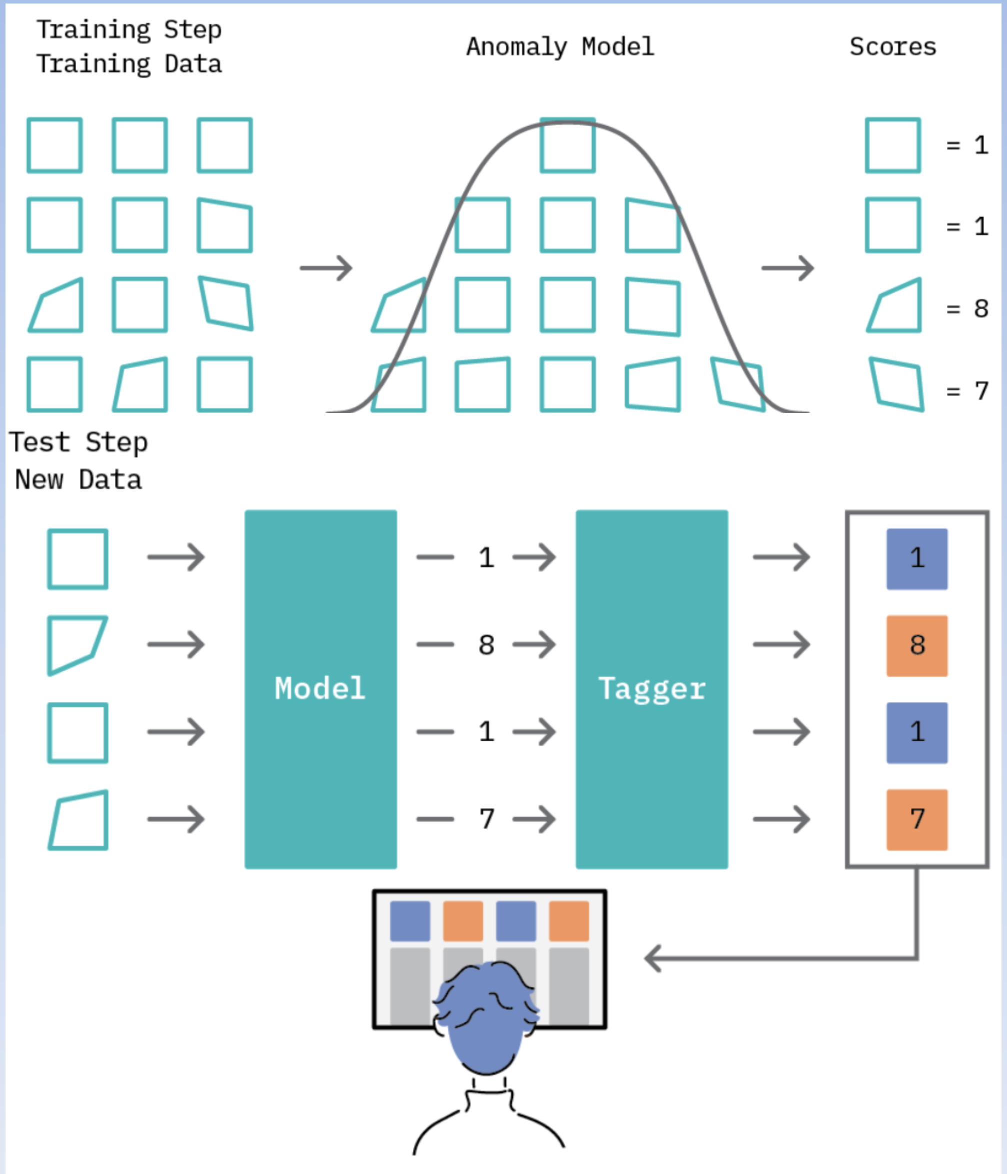
1.Construct an Isolation Tree either from the entire feature set or a randomly chosen subset of the feature set.

2.Construct n such Isolation trees.

3.Calculate an **Anomaly score** for each datapoint.

- The **Anomaly score** is a non-linear function of the Average path length over all Isolation trees.
- The path length is equivalent to the number of splits made by the Isolation tree to isolate a point.
- The shorter the Average path length, the larger are the chances of the point being an anomaly(as we saw earlier in the diagram).

Machine Learning in Anomaly Detection



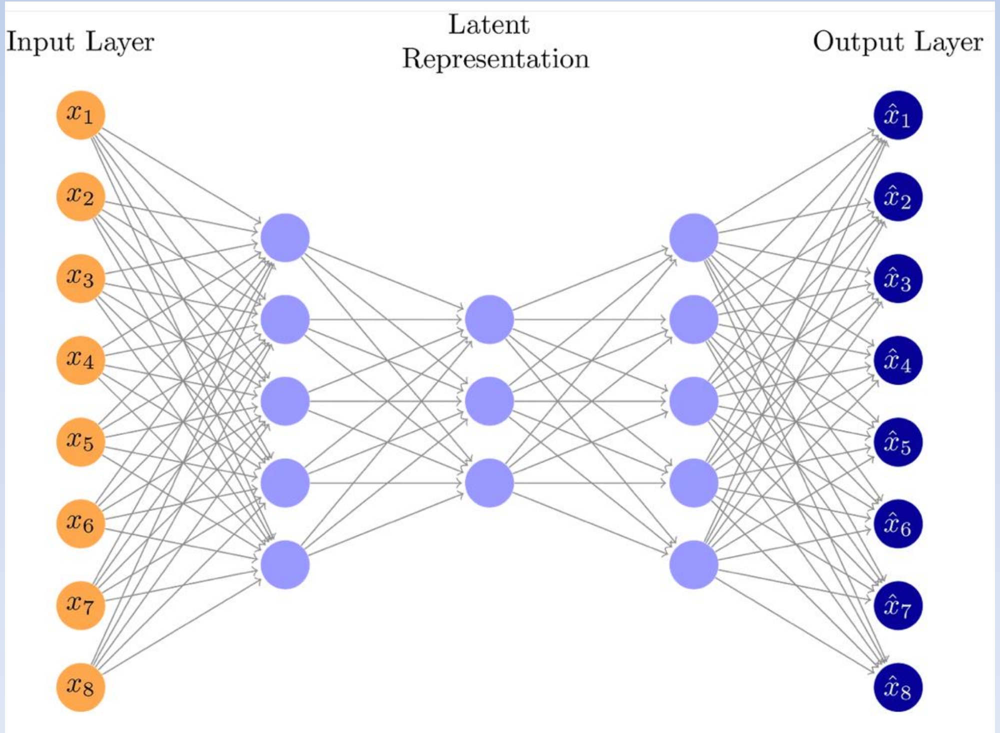
Machine learning methods are among the most used for Anomaly Detection

The first step, referred to as the training step, involves building a model of normal behavior using available data

Based on this model, an anomaly score is then assigned to each data point that represents a measure of deviation from normal behavior.

Autoencoders

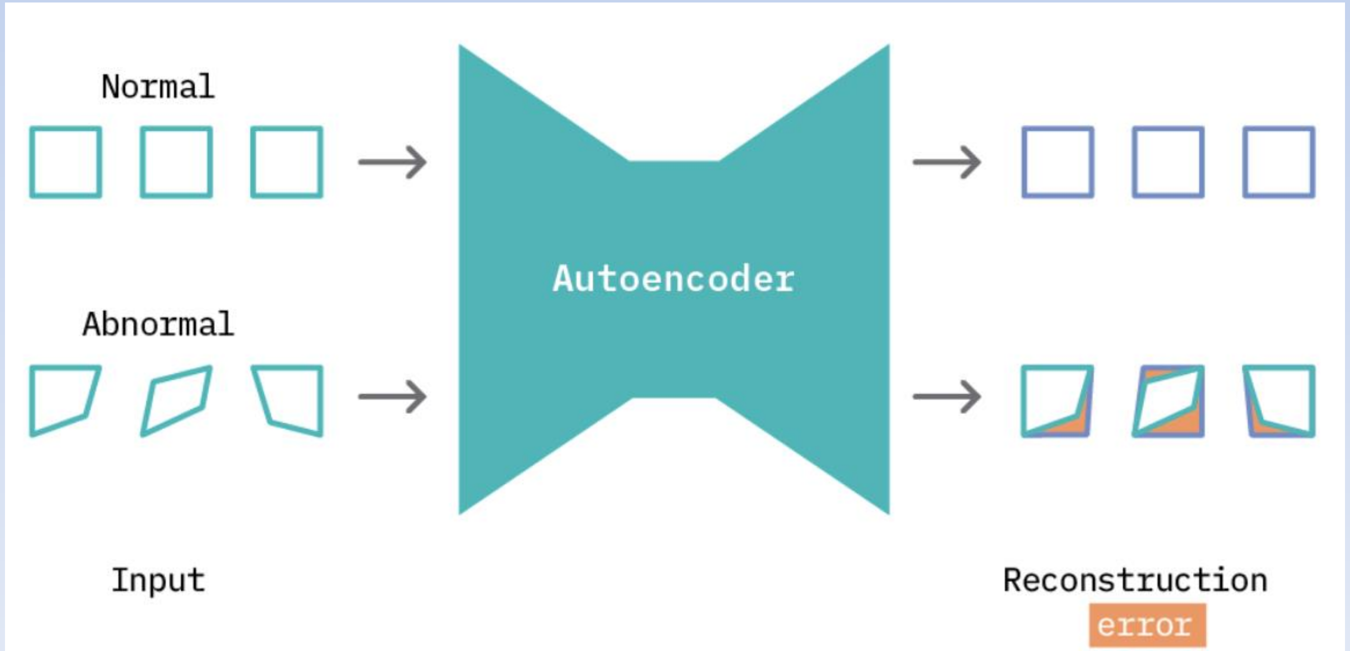
- Autoencoders are neural networks designed to learn a low-dimensional representation, given some input data.
- They consist of two components: an encoder that learns to map input data to a low-dimensional representation (latent representation), and a decoder that learns to map this low-dimensional representation back to the original input data.



General Autoencoder

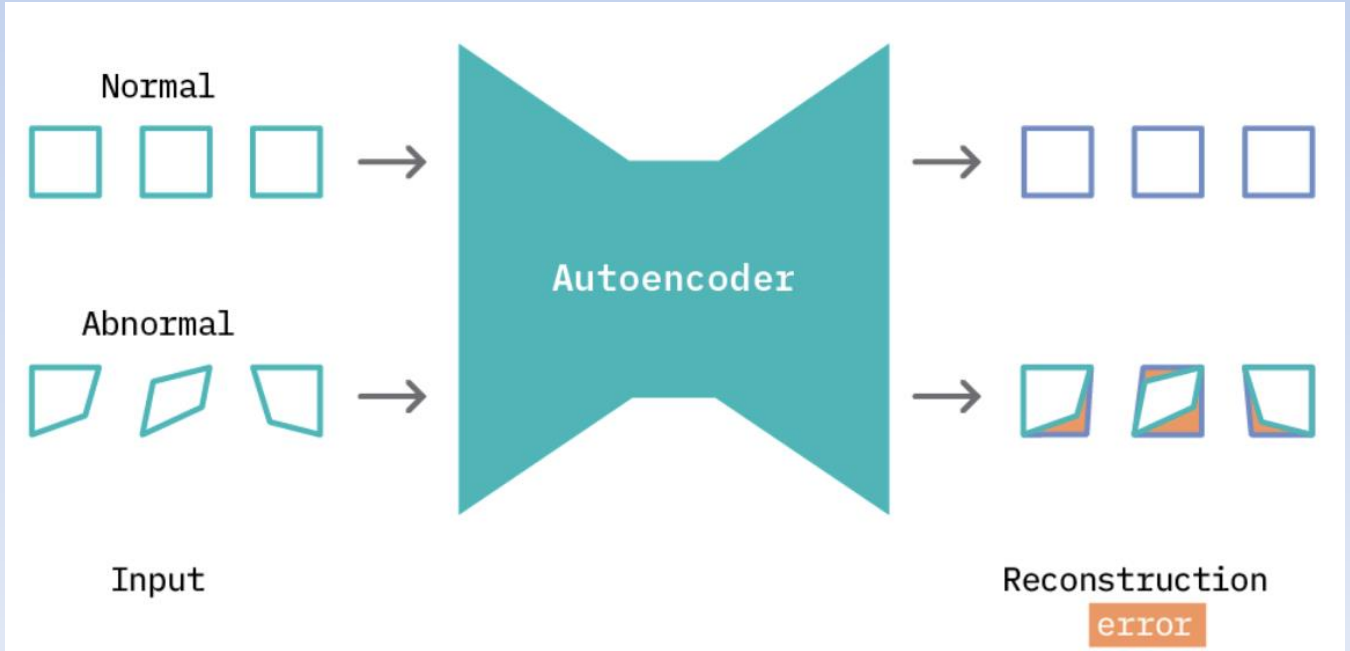
Autoencoders

- Applying an autoencoder for anomaly detection follows the general principle of first modeling normal behavior and subsequently generating an anomaly score for each new data sample.
- To model normal behavior, we follow a semi-supervised approach where we train the autoencoder on normal data samples. This way, the model learns a mapping function that successfully reconstructs normal data samples with a very small reconstruction error.



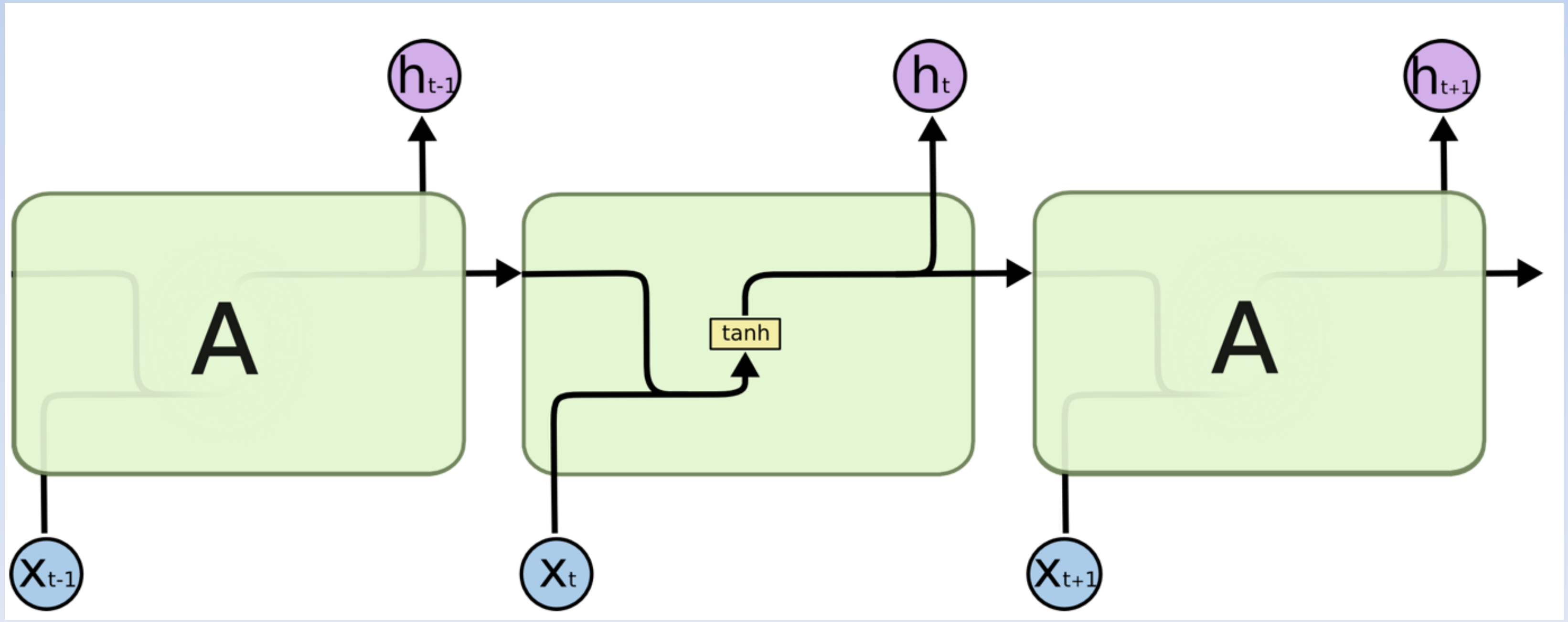
Autoencoders

- This behavior is replicated at test time, where the reconstruction error is small for normal data samples, and large for abnormal data samples.
- To identify anomalies, we use the reconstruction error score as an anomaly score and flag samples with reconstruction errors above a given threshold.
- .



LSTMs

- Long Short Term Memory networks (LSTMs) are a special kind of recurrent neural network (RNN) capable of learning long-term dependencies in sequence data.
- LSTMs were created to overcome the inability to retain information for long periods of time, which is an inherent limitation in RNN
-



Discrete – Time Markov Chains

• Many real-world systems contain uncertainty and evolve over time.

• Stochastic processes (and Markov chains) are probability models for such systems.

• A discrete-time stochastic process
is a sequence of random variables
$X_0, X_1, X_2, \ldots$ typically denoted by $\{ X_n \}$.

• Origins: Galton-Watson process → When and with what probability will a family name become extinct?

Components of Stochastic Processes

The state space of a stochastic process is
the set of all values that the X_n's can take.

(we will be concerned with
stochastic processes with a finite # of states)

Time: $n = 0, 1, 2, \ldots$

State: v-dimensional vector, $\mathbf{s} = (s_1, s_2, \ldots, s_v)$

In general, there are m states,

$\mathbf{s}^1, \mathbf{s}^2, \ldots, \mathbf{s}^m$ or $\mathbf{s}^0, \mathbf{s}^1, \ldots, \mathbf{s}^{m-1}$

Also, X_n takes one of m values, so $X_n \leftrightarrow \mathbf{s}$.

Gambler's Ruin

At time 0, I have ***X_0 = 2*** EUR, and each day I make a 1 EUR bet. I win with probability p and lose with probability $1 - p$. I'll quit if I ever obtain 4 EUR or if I lose all my money.

State space is ***S*** = { 0, 1, 2, 3, 4 }

Let X_n = amount of money I have after the bet on day n.

X1 = 3 with probability p and 1 with probability 1-p

If ***Xn = 4***, then $X_{n+1} = X_{n+2} = \cdots = 4$.

If ***Xn = 0***, then $X_{n+1} = X_{n+2} = \cdots = 0$.

Markov Chain Definition

A stochastic process $\{ X_n \}$ is called a Markov chain if

$$\Pr\left\{ X_{n+1} = j \mid X_0 = k_0, \ldots, X_{n-1} = k_{n-1}, X_n = i \right\}$$

$$= \Pr\left\{ X_{n+1} = j \mid X_n = i \right\} \quad \leftarrow \text{transition probabilities}$$

for every $i, j, k_0, \ldots, k_{n-1}$ and for every n.

Discrete time means $n \in N = \{0, 1, 2, \ldots\}$.

The future behavior of the system depends only on the current state *i* and not on any of the previous states.

Stationary Transition Probabilities

$$\Pr\{ X_{n+1} = j \mid X_n = i \} = \Pr\{ X_1 = j \mid X_0 = i \} \text{ for all } n$$

(They don’t change over time)

We will only consider stationary Markov chains.

The one-step transition matrix for a Markov chain with states $\boldsymbol{S} = \{ 0, 1, 2, \ldots N \}$ is

$$P = \begin{pmatrix} p_{1,1} & p_{1,2} & \cdots & p_{1,N} \\ p_{2,1} & p_{2,2} & \cdots & p_{2,N} \\ \vdots & \vdots & \vdots & \vdots \\ p_{N,1} & p_{N,2} & \cdots & p_{N,N} \end{pmatrix}$$

where $p_{ij} = \Pr\{ X_1 = j \mid X_0 = i \}$

Properties of Transition Matrix

If the state space $\boldsymbol{S} = \{0, 1, \ldots, m-1\}$ then we have

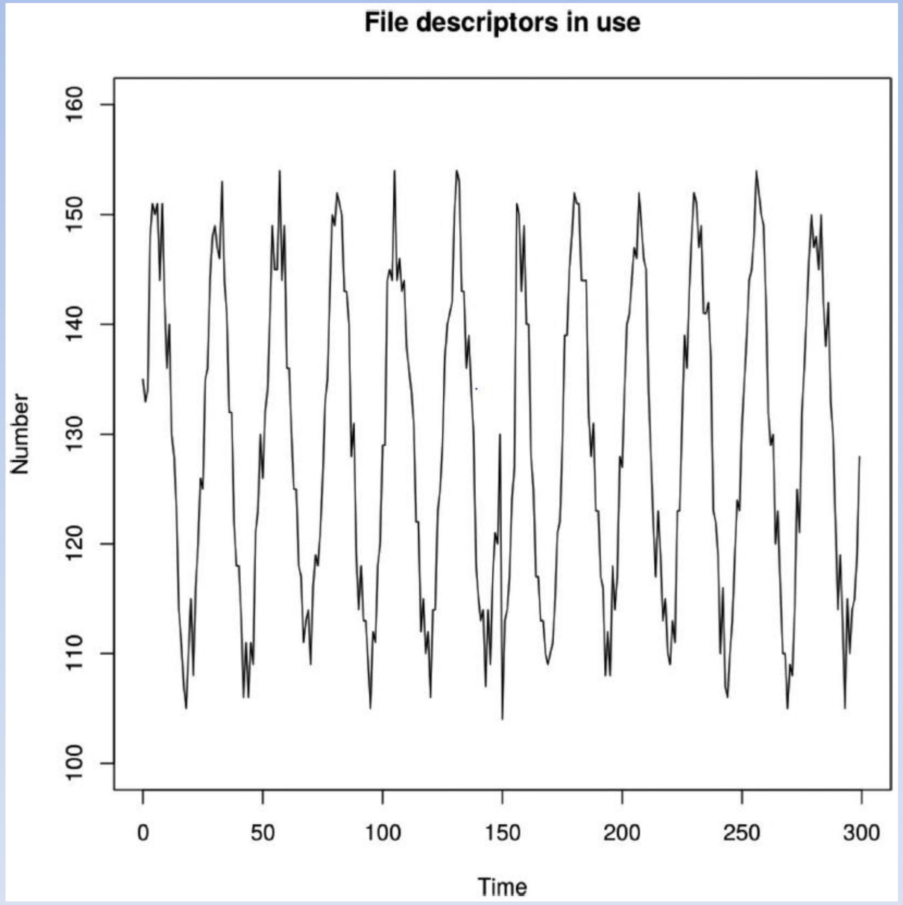
$\sum_j p_{ij} = 1 \ \ \forall\, i$ and $p_{ij} \geq 0 \ \ \forall\, i, j$

(we must go somewhere) (each transition has probability ≥ 0)

Gambler's Ruin Example

$$\begin{array}{c|ccccc}
 & 0 & 1 & 2 & 3 & 4 \\
\hline
0 & 1 & 0 & 0 & 0 & 0 \\
1 & 1\text{-}p & 0 & p & 0 & 0 \\
2 & 0 & 1\text{-}p & 0 & p & 0 \\
3 & 0 & 0 & 1\text{-}p & 0 & p \\
4 & 0 & 0 & 0 & 0 & 1
\end{array}$$

System States

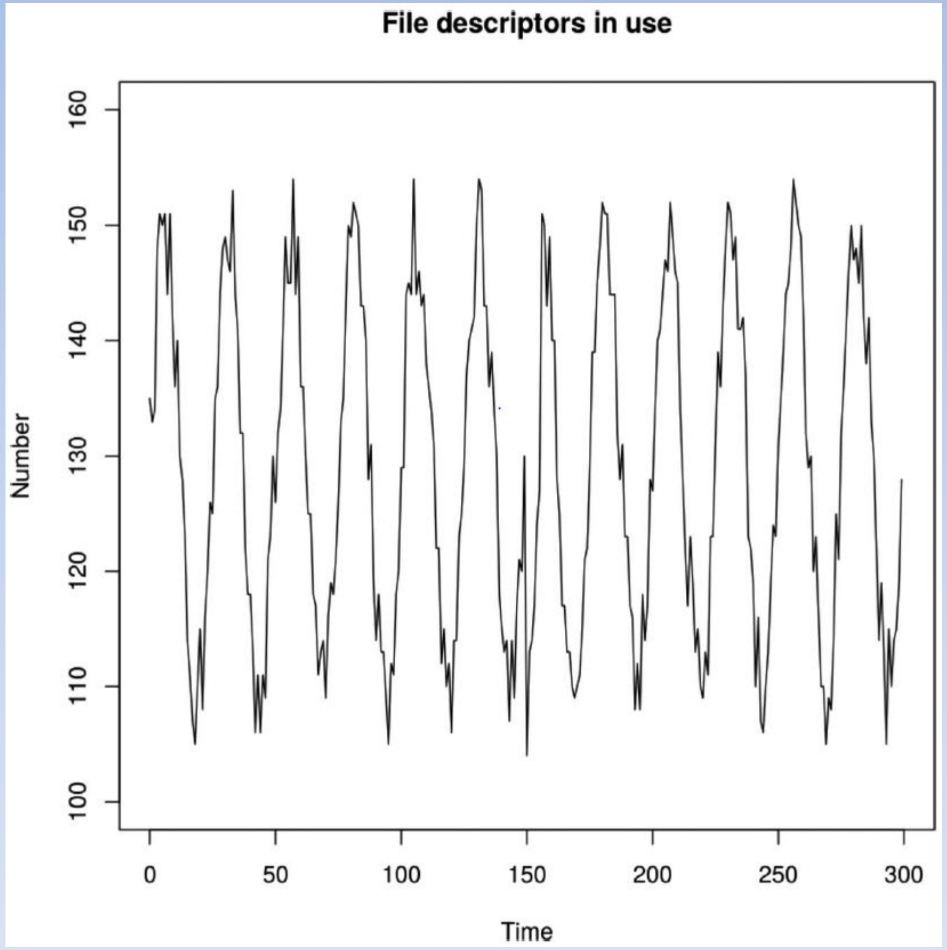


The first step in representing a system as a Markov chain is to itemise its possible states. For the file system example an obvious, **but incorrect**, way of representing the state at any time is by the number of file descriptors in use

While this is simple, it unfortunately breaks the fundamental assumption of Markov chains :

> *the probability of being in state j at time t + 1 depends only on being in state i at time t. The history of how state i was reached is irrelevant*

System States



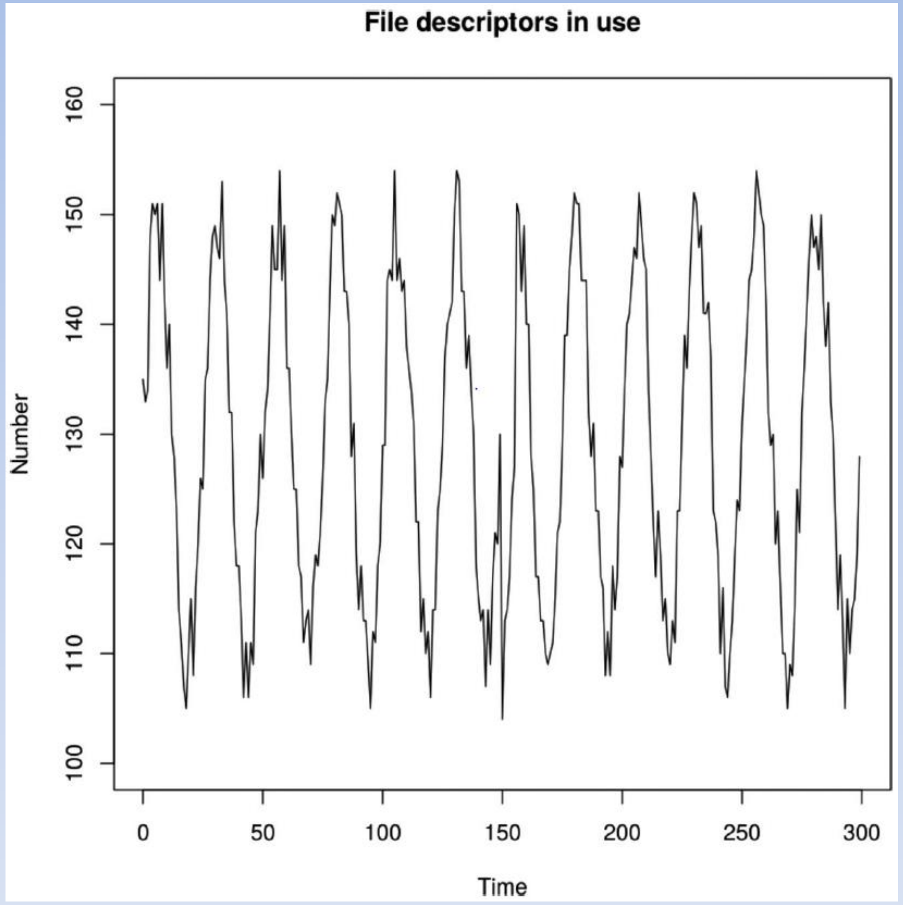
Le us assume that the current number of le descriptors in use is 131 and that we want to predict how many will be in use at the next sampling time

To apply a Markov chain, we must assume that this will not depend on any earlier values

this is not true, because the use of descriptors is clearly cyclic, and the value of 131 can occur both on a downward and an upward part of the cycle.

During a downward cycle, the probability of moving from 131 descriptors in use to 129 in use is much higher than it would be on an upward cycle

System States



To avoid this, our representation of the system state will have to include details of whether the system is on the downward or upward part of the cycle: 131U, 131D

In principle, it would then be possible to draw a diagram showing the possible transitions between states and their likelihood

Using the learning, there would be about 100 states and 2014 arrows between them

Learning the Transition Probabilities

- In order to decide when something is anomalous, we need to know what is normal
- For a Markov chain, we build the probability of transitions between states by observing normal behavior, possibly during system testing, possibly during field trials. This allows us to build a matrix of transition probabilities
- Training consists of observing the system and determining what proportion of the transitions out of ***state i*** are moves to ***state j***.
- In the case of our example, 10,000 readings where observed and the transition probabilities calculated

Learning the Transition Probabilities

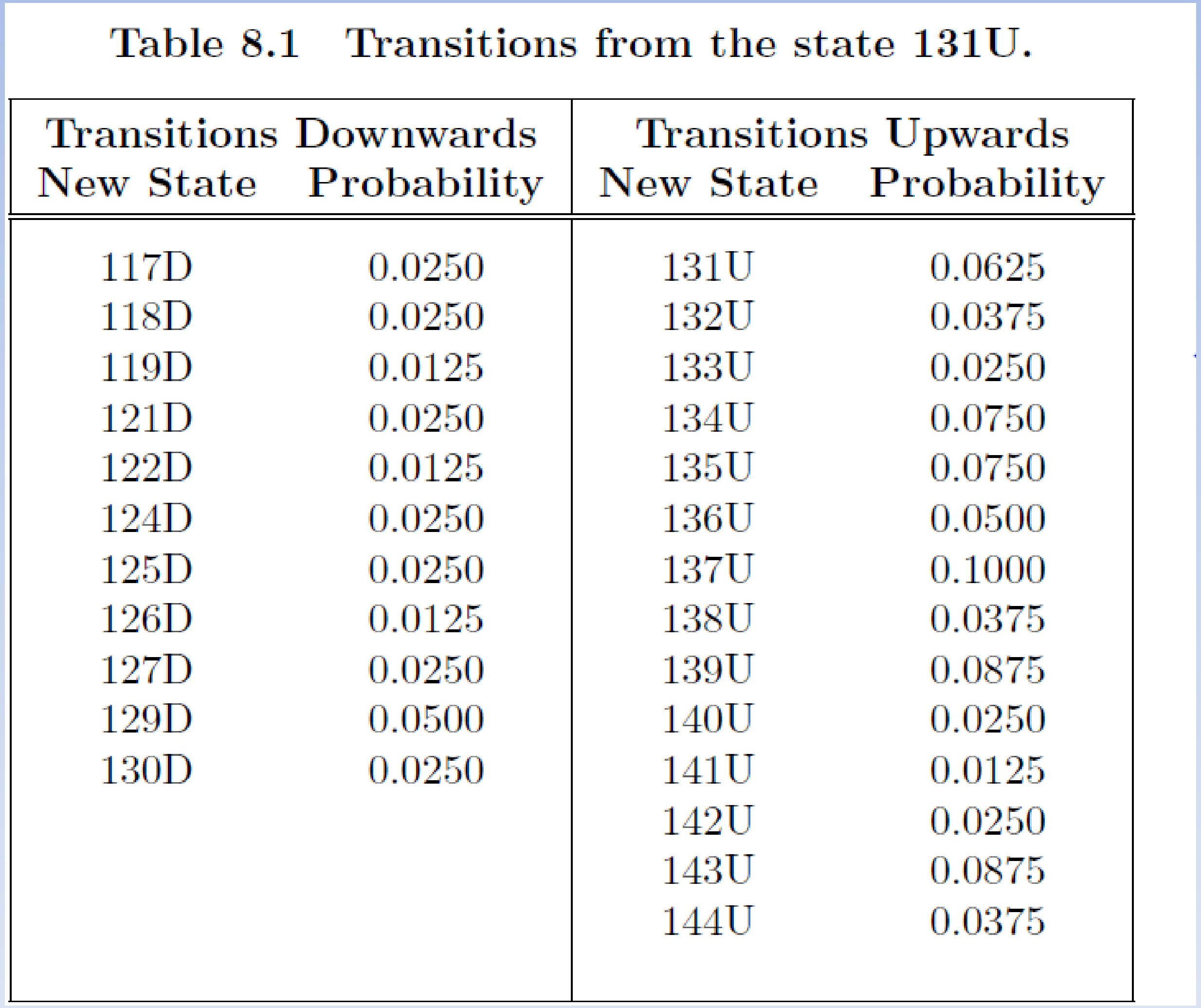


Table 8.1 Transitions from the state 131U.

Transitions Downwards		Transitions Upwards	
New State	Probability	New State	Probability
117D	0.0250	131U	0.0625
118D	0.0250	132U	0.0375
119D	0.0125	133U	0.0250
121D	0.0250	134U	0.0750
122D	0.0125	135U	0.0750
124D	0.0250	136U	0.0500
125D	0.0250	137U	0.1000
126D	0.0125	138U	0.0375
127D	0.0250	139U	0.0875
129D	0.0500	140U	0.0250
130D	0.0250	141U	0.0125
		142U	0.0250
		143U	0.0875
		144U	0.0375

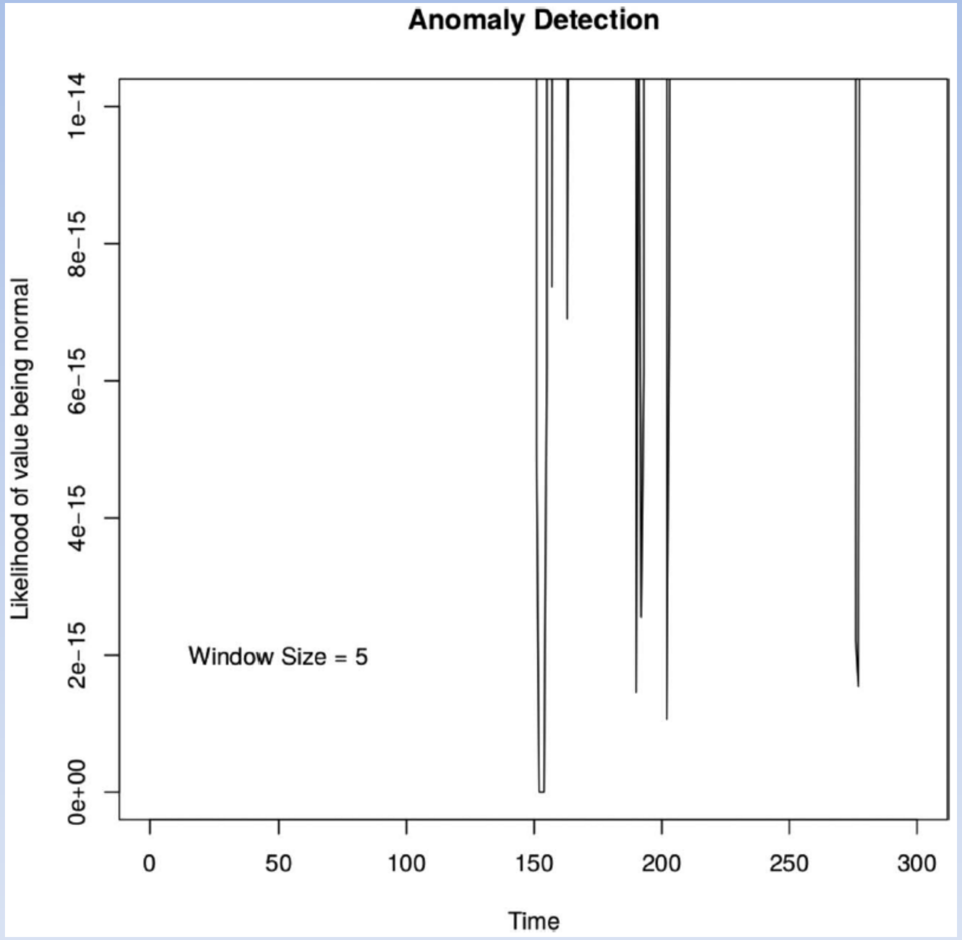
Looking for an Anomaly

- The objective is to look at a series of transitions in the system states and determine, given the transition probability matrix how likely it is for that series to have occurred in normal operation
- Given a “window" of P transitions, the likelihood is calculated as:

$$L = \prod_{t=1}^{P-1} P(X_{t-1}, X_t)$$

- Where P(a, b) is the probabilit tate a to state b

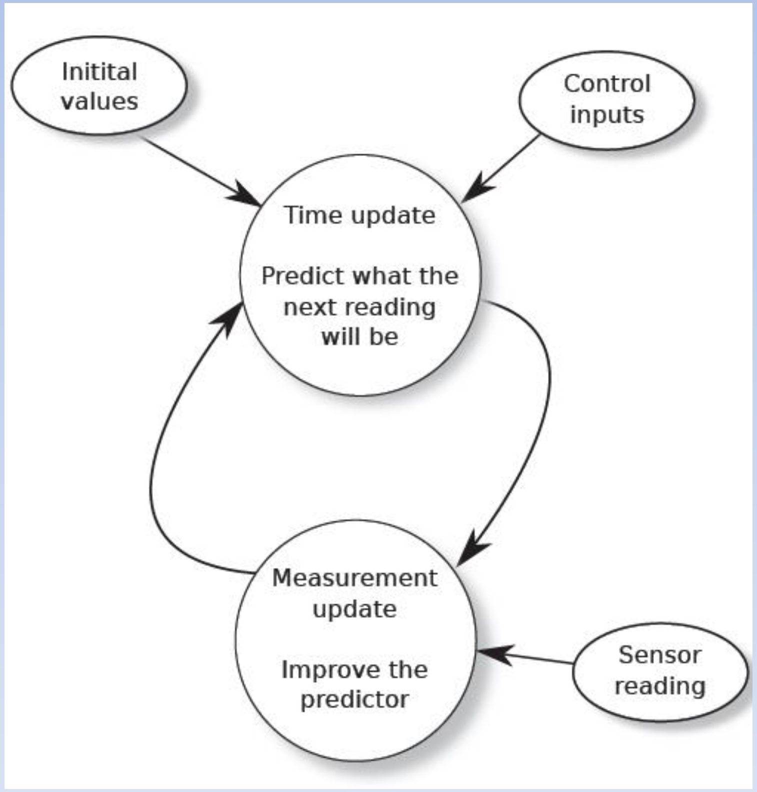
Looking for an Anomaly



- Some experimentation is needed to determine the window size the length of the sequence to be observed
- The figure displays the analysis on the number of file descriptors in use for a window size of 5
- The vertical axis is a measure of how likely it is that the series of 5 states would have occurred during normal operation, without an anomaly being present
- It can be seen that the measure of likelihood of the sequence of transitions following time t = 150 is effectively zero: there is clearly an anomaly at that time.
- The computation also finds anomalies around time 200 and 260

Kalman Filters

- A Kalman filter is a predictor/corrector algorithm

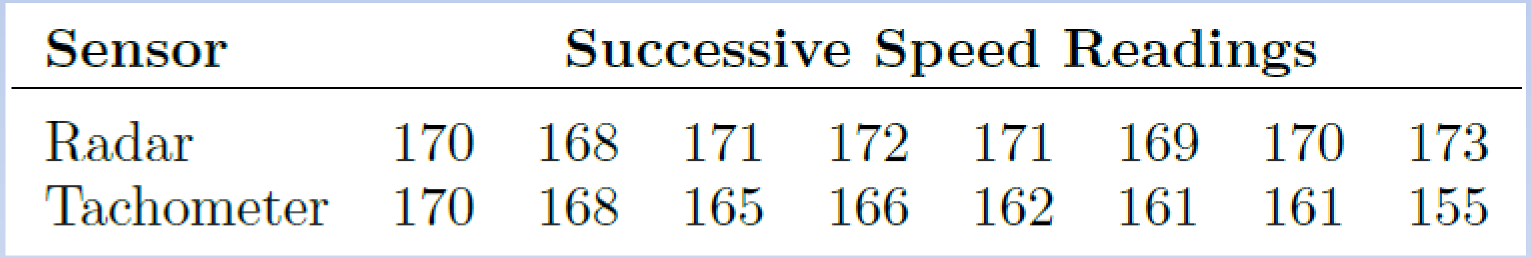


Kalman Filter

- At time T, the algorithm predicts the value that will be received from the sensors at time T + 1, and, when the actual reading is received, it adjusts the predictor to make it more accurate next time
- Predicting what the next reading will be is useful for anomaly detection because, once the algorithm has settled down and is accurate in its predictions, a major difference between the predicted and actual values represents a potential anomaly

Kalman Filter

- Assume that the state of the system can be described by a vector, **x** of n values (for the example n=1)
- If we had two sensors:



Sensor	Successive Speed Readings							
Radar	170	168	171	172	171	169	170	173
Tachometer	170	168	165	166	162	161	161	155

- Neither sensor by itself appears to be anomalous; what is anomalous is that one sensor is detecting a deceleration, while the other is not

Kalman Filter

- A Kalman filter predicts the future value of **x** if the system can be described at the time k by an equation of the form

$$\vec{x}_k = A\vec{x}_{k-1} + B\vec{u}_{k-1} + \vec{w}_{k-1}$$

- and measurements it receives (sensor values)

$$\vec{z}_k = H\vec{x}_k + \vec{v}_k$$

Kalman Filter

- is the sta$\vec{x}_k$ of the system at time *k*. This is unknown to the software, which knows only the noisy measurements that it is receiving from the sensors
- A is an *n*x*n* matrix that relates to according to the physics of the problem, assuming that there is no noise on the ser$\vec{x}_{k+1}$ rea$\vec{x}_k$ngs
- B is an *n*x*l* matrix that relates one or more control inputs to the new values of
- is a vector of length *l* that relates the control input specified by B to the expected change

$\vec{x}_k$

$\vec{u}_k$

Kalman Filter

- is the n$\vec{w}_k$e in the process itself (not the noise in the measurements). For many systems, the process itself is not noisy (is a vector set to zero values)
- is the measurement taken from the m sensors
- H is an *mx*$\vec{z}_k \in \mathbb{R}^m$ that relates the actual system state to a measurement. If there is a single sensor, then H would be a single number, normally with value 1
- is the noise in the measurements

$\vec{v}_k$

Kalman Filter: Prediction

- Given the symbols listed above, the prediction and correction equations are as follows:

$$\vec{x}_k = A\vec{x}_{k-1} + Bu_{k-1}$$

$$P_k = AP_{k-1}A^T + Q$$

- P is the covariance matrix of the errors and can be initialized to any nonzero value as it will iterate to a correct value over time.
- Q is the process noise covariance and can normally be set to something very small (a value of 0 should be avoided, but for a one-dimensional model, a value such as Q = 10^{-5} would not be unreasonable)

Kalman Filter:Correction

- The * indicates the predicted values

$$K_k = P_k^* H^T (H P_k^* H^T + R)^{-1}$$

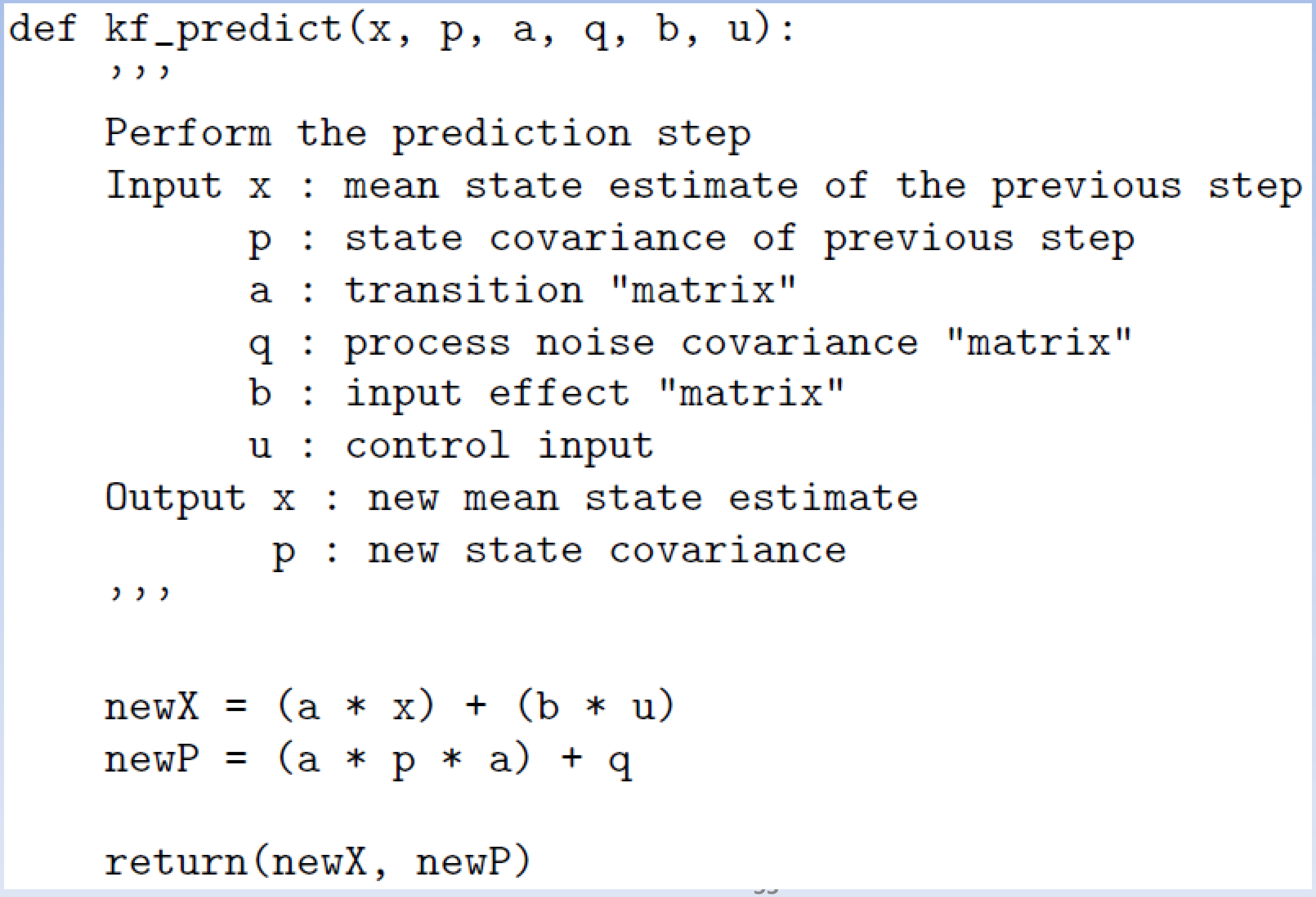
$$\vec{x}_k = \vec{x}_k^* + K_k (z_k - H \vec{x}_k^*)$$

$$P_k = (I - K_k H) P_k^*$$

- A Kalman filter can be used to detect anomalous values read from the sensors. A predicted value for the next reading can be compared with the actual reading when it arrives and the difference calculated

$$\vec{v}_k = \vec{z}_k - H \vec{x}_k^*$$

Kalman Filter Code



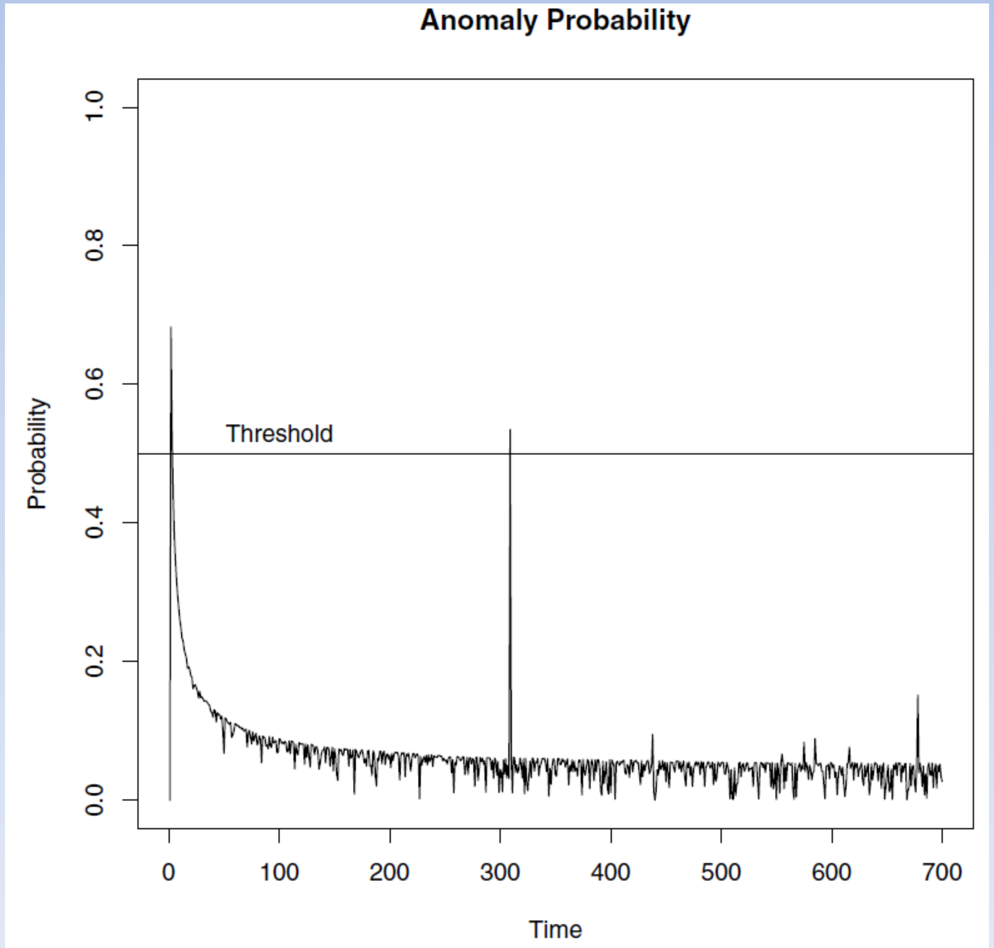
```
def kf_predict(x, p, a, q, b, u):
    '''
    Perform the prediction step
    Input x : mean state estimate of the previous step
          p : state covariance of previous step
          a : transition "matrix"
          q : process noise covariance "matrix"
          b : input effect "matrix"
          u : control input
    Output x : new mean state estimate
           p : new state covariance
    '''

    newX = (a * x) + (b * u)
    newP = (a * p * a) + q

    return(newX, newP)
```

Train Example

- The figure illustrates the probability of anomaly of the values given



Looking for an Anomaly

- As has been stated above, this is a particularly simple application of the Kalman filter because it is one-dimensional and the only value of interest is the train's speed. This means that all the vectors and matrices are numbers and the code to perform the algorithm is extremely simple.
- Once the system has settled down and the filter is able to make reasonable predictions about the next value, the probability of anomaly remains generally low, apart from the value given at time t = 308, which we know from the figure that is anomalous
- It is interesting that the other value that appears to be anomalous when t = 437, appears as a spike in the graph

Diseño de Software Crítico

- Patrones para gestión de errores
  - Detección de anomalías: cadenas de Markov y filtros de Kalman
  - Manejo de errores: Rejuvenation y Bloques de Recuperación
- Replicación y diversificación
- Balances arquitectónicos: disponibilidad/fiabilidad, seguridad/prestaciones,…

Error Handling: Rejuvenation

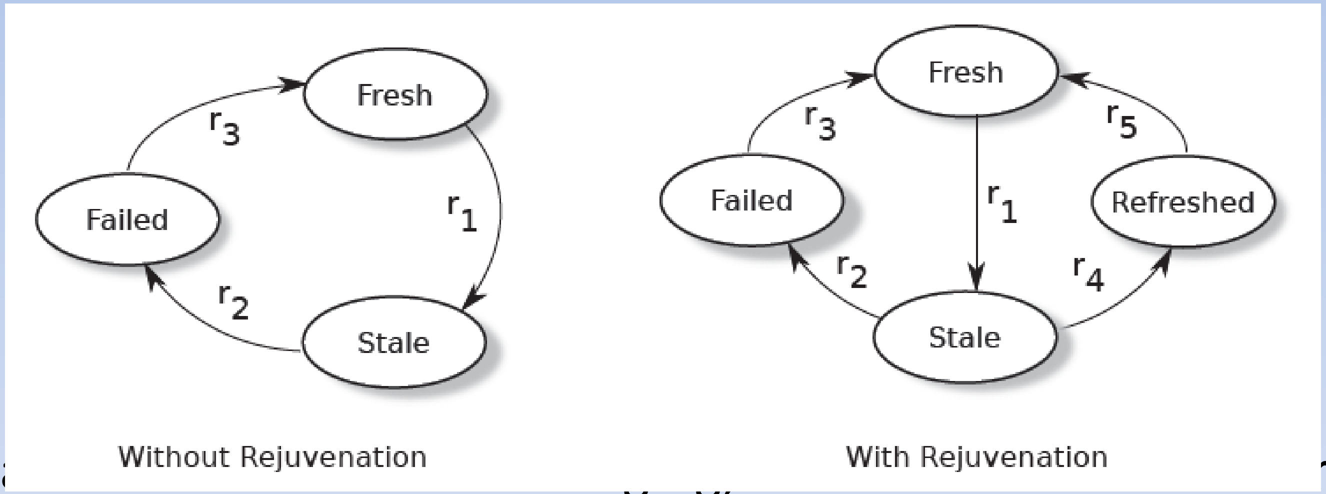
- Our purpose is to prevent an error from becoming a failure or, if that is not possible, to prepare for the failure in as controlled a manner as possible
- Rejuvenation is one means of breaking the link between error and failure in a controlled manner
- The term “rejuvenation" is just the formal way of saying, “press the RESET button from time to time“
- The purpose behind rejuvenation is to intercept and remove errors before they become failures
- Faults in the code may have caused a number of errors (e.g. zombie threads, data fragmentation and Heisenbug corruptions of memory, resource leakage, etc) that have occurred but have not yet caused failures

Rejuvenation: When

- Most embedded systems have a natural operating periodicity
  - An aircraft flies for no more than 36 hours
  - a medical device is in continuous use for no more than a month
  - a car is driven for no more than a few hours, etc
- Given this periodicity, it is unclear why a reboot should not be built into the operating requirements at these convenient times
  - For a designer, demanding a periodic rebooting of the system is considered by many to be failure in some way

Rejuvenation: What

- Viewed from an availability analyst's point of view, rejuvenation is a form of periodic replacement, such as occurs in hardware maintenance



Without Rejuvenation With Rejuvenation

- Stale state is a st red) at a rate of r_1 per hour
- Once in the stale state, there is a rate, r_2, at which the system moves into its failure state (r_3 is the repair rate)

Rejuvenation

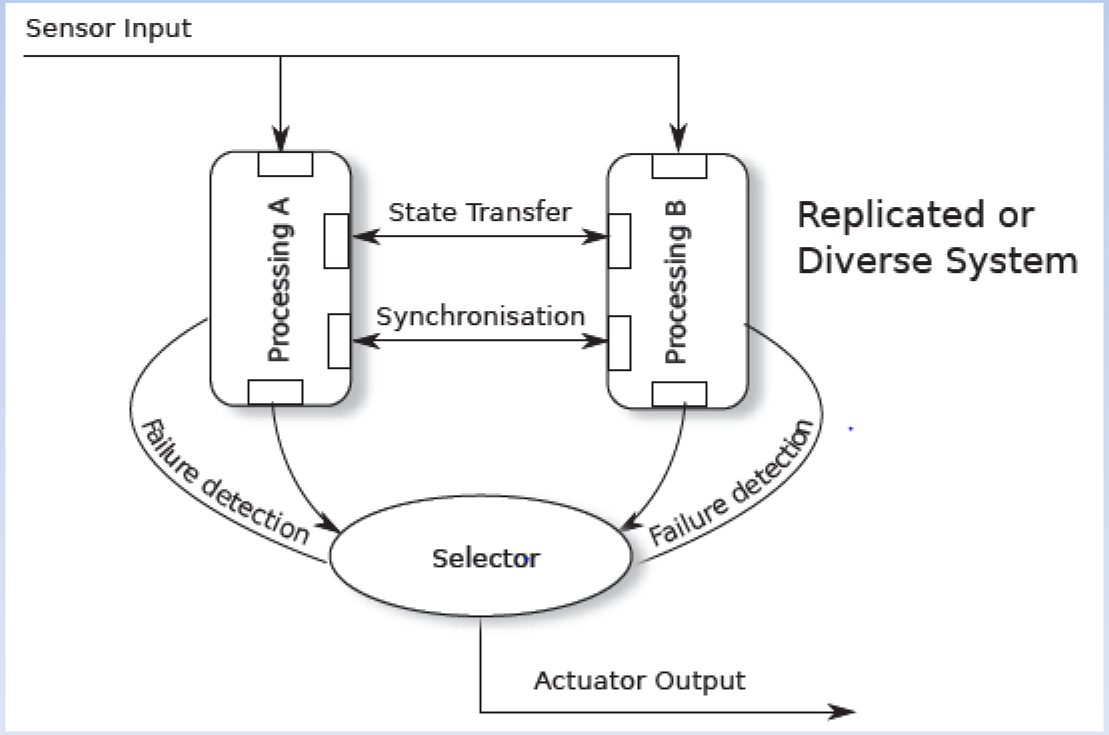
- When the system is stale, rejuvenation occurs at a rate r_4, and this can avoid the move to the failed state.
- The main problems associated with rejuvenation are determining the correct rejuvenation rate
- If r_4 is too large, representing very frequent pressing of the reset button, then system availability drops
- If r_4 is too small the technique has limited use, because the r_2 transition can dominate when the system is stale, causing a failure and again impacting the availability of the system

Rejuvenation

- Selecting r_4 :
  - If the system has an operating time $1/r >> 1/r_2$ then rejuvenation can take place during its maintenance period
  - this may be valid for systems in aircraft, cars, railway trains, medical devices, and other systems regularly taken out of service
  - In this case, it is essential that the maximum rejuvenation interval be explicit in the safety manual and be included as part of the claim in the safety case
  - In other cases, the rejuvenation may need to be triggered by an anomaly detection system

Rejuvenation in Replicated Systems

- In cases where a replicated or diverse design has been employed rejuvenation may be useful not only to cleanse the operating (hot) subsystem, but also to exercise the changeover mechanism and ensure that the cold (non operating) subsystem has not failed silently



Rejuvenation in Replicated Systems

- If this is done before the hot subsystem has failed, a rapid switch may still be possible back to the original subsystem if it is found that the standby has failed silently
- In this model, rejuvenation may be triggered by time (invoking the switch-over every X hours) or whenever an anomaly is detected in the hot (operating) subsystem
- IEC 61508-2, the hardware section of IEC 61508, refers to the concept of “proof testing" for hardware

> **proof test: periodic test performed to detect dangerous hidden failures in a safety-related system so that, if necessary, a repair can restore the system to an “as new” condition or as close as practical to this condition**

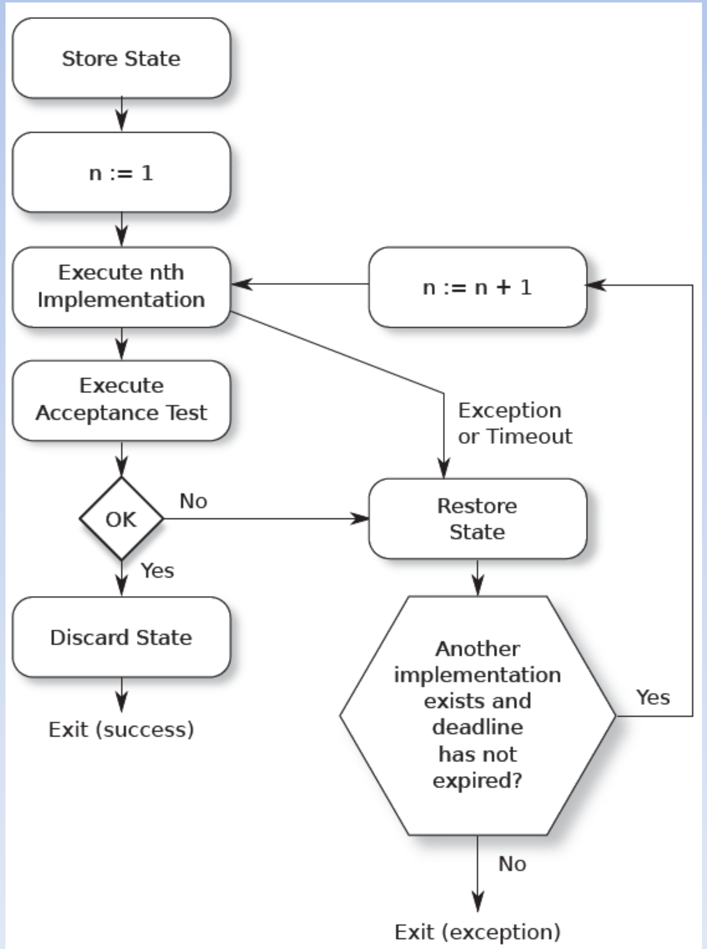
Recovery Blocks

- Another particularly simple (and effective) technique for detecting an error and recovering is that of “recovery blocks“
  - The term “block” reflects the origin of the method back in the ALGOL language that was in use in the 1970s
- Recovery blocks are explicitly mentioned in Table 5 of ISO 26262-6 as a “Mechanism for error handling at the software architectural level.“
- The basic pattern of a recovery block is that, instead of calling a particular function to carry out a computation, we instead structure the code as follows:

```
ensure <acceptance test> by <function1>
else by <function2>
...
else by <functionN>
else error
```

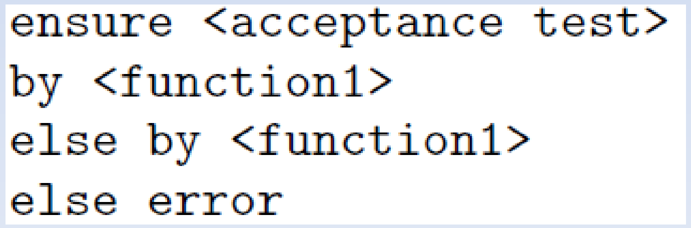
Recovery Blocks

- The acceptance test is a routine that evaluates the result returned by the function and indicates whether it is acceptable or not in the context



Recovery Blocks

- There are many variants of this basic technique, which can be thought of as a version of N-version programming
  - One variant stems from the observation that the different implementations of the function do not need to calculate identical results, they only need to return acceptable results
  - So function1 might try to handle the input parameters fully, whereas function2 might perform a simpler computation giving a more approximate answer.
  - A second observation arises from the fact that most faults remaining in shipped code are likely to be Heisenbugs. If this is the case then the pattern can be simplified:



```
ensure <acceptance test>
by <function1>
else by <function1>
else error
```

Recovery Blocks

- One disadvantage of recovery blocks is that any side effects (system changes) made by a routine whose output is eventually rejected by the acceptance routine must be undone before the computation is repeated
  - These side effects include “information smuggling” when the routine changes data structures outside the group of programs engaged in the processing or even dynamically creates new proceses
- Recovery blocks can also provide some level of protection against security attacks, because the attacker has to corrupt both the implementation and the acceptance test before a bad value can be silently generated

Diseño de Software Crítico

- Patrones para gestión de errores
  - Detección de anomalías: cadenas de Markov y filtros de Kalman
  - Manejo de errores: Rejuvenation y Bloques de Recuperación
- Replicación y diversificación
- Balances arquitectónicos: disponibilidad/fiabilidad, seguridad/prestaciones,…

Expecting the Unexpected
Design Safe State

- For a system with safety implications, it is essential to acknowledge that it will sometimes encounter conditions that it was not designed to handle
- These conditions may occur because of random corruption caused by hardware or software (Heisenbugs) or may simply be a circumstance unforeseen in the design that leads to a deadlock or livelock
- The behavior of the system under these conditions must be defined and must be included in the accompanying safety manual so that the designer of any larger system into which this one is incorporated can detect the condition and take the appropriate action

Expecting the Unexpected
Design Safe State

- The action to be taken is to move, if possible, to the system's currently applicable “design safe state" defined during the system design
- For some systems, the design safe state may be obvious and unchanging over time
- For example:
  - traffic lights at a busy road intersection might all be set to red,
  - a system that controls the brakes on a train might always apply the brakes

Expecting the Unexpected
Design Safe State

- For other systems, the design safe state may not be so obvious and may vary from situation to situation
  - For a medical device that dispenses drugs, is it safer to continue drug administration when the monitoring system meets an unexpected condition or to stop the drug flow?
  - For a car's instrument cluster, if the software in the graphics system meets an unanticipated situation, is it best to blank the screen, to freeze the information that is on the screen, or to display sane but incorrect information?
  - For a system that controls the braking of a car, perhaps part of a collision detection system. When the car is travelling at 10 km/hr on a deserted side road, a suitable design safe state might be for the failing system to apply the brakes. At 120 km/hr on a highway, suddenly applying the brakes might be exactly the wrong thing to do.

Design Safe State

- Whatever the design safe state, it must have three characteristics:
  1. It must be externally visible in an unambiguous manner
  2. It must be entered within a predefined, and published, time of the condition being detected
  3. It must be documented in the safety manual so that a designer using the component can incorporate it into a larger design
- It is also useful if entering the design safe state causes some form of diagnostics to be stored so that the condition can be investigated

Design Safe State

- However well the design safe state is defined, it must also be accepted that it will sometimes be impossible for the system to reach it
  - The internal corruption that has caused the unexpected condition may be so severe as to prevent the system from taking any form of coherent action.
  - There is always some level of hardware that must behave properly to allow even a simple software process to execute
- Knowing the design safe states of components allows the designer who incorporates those components into a larger system to plan accordingly
- If moving into the design safe state causes the system to discard input already received and assumed processed, then this must also be defined in the safety manual so that the designer of the larger system can handle the condition

Design Safe State Standards

- ISO 26262-3 addresses the system's safe state including the statement that, “If a safe state cannot be reached by a transition within an acceptable time interval, an emergency operation shall be specified.“
- ISO 26262 also specify designers that the system must not only be able to move to its safe state, but mechanisms must also be in place to hold it there until external action is taken
- IEC 61508 also has many references to the design safe state

Design safe state: In certain circumstances, upon controlled failure of the system application, the element may revert to a design safe state. In such circumstances, the precise definition of design safe state should be specified for consideration by the integrator.

Design Safe State and Dangereous Failure

- The purpose of the design safe state is to provide the designer with a way of handling unexpected conditions
- The question is often asked whether moving to the design safe state constitutes a dangerous failure and should therefore be considered in the calculation of the failure rate for IEC 61508's safety integrity level (SIL)
- To meet the requirements of SIL 3, for example, a dangerous failure rate of $< 10^{-7}$ failures per hour of operation has to be demonstrated

If a system is so poorly designed that, on average, it meets an unexpected condition every 1000 hours, but always detects that condition and moves to its design safe state, then can that system be considered to meet the requirements of SIL 3?

Design Safe State and Dangereous Failure

- This is a question of the usefulness/safety balance, to be discussed later on software architecture balance
  - It can be argued that, almost by definition, a move to the design safe state is not a dangerous failure. **Why?**
  - include it where possible in the IEC 61508 calculation because, even though the larger system should be designed to handle a component moving to its design safe state, such moves will always place stress on the larger system and move it closer to a dangerous condition

Recovery

- When an unanticipated condition occurs, there are two possible actions:
  - to attempt to recover, for example, by using recovery blocks
  - to move directly to the design safe state
- Software-based recovery often takes the shape of executing scripts from a "recovery tree."

If process X reports that it has met an unexpected situation, then kill processes X, Y, and Z. Then restart Z, wait 200ms, restart Y, wait until ag A has been set, and
then restart X

- The intent is to try to preserve as much of the state of the system as possible, to restrict the corruption that could be caused by invalid information being transferred to another part of the system, and then to restart and resynchronize the affected subsystems with as little system level disruption as possible

Do you want to recover?

- By definition, any attempt to recover from an unanticipated situation means executing an algorithm that has not been tested, and this execution will happen at a time when the system state is ill-defined
  - For many systems it can be argued that executing untested code that implements an untested algorithm on a system whose state is ill-defined and certainly unexpected, is not suitable for a safety-critical device

- Quite often such heroic recovery attempts to avoid moving to the design safe state are only partially successful, leaving the system in a worse condition than it might have been in before.

Crash only model

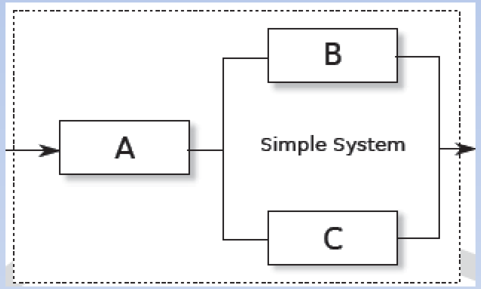
- The crash-only model is self-explanatory: on detecting an unanticipated condition, the system crashes and stops any form of further processing
- Advantages:
  - It relies on only a small part of the system continuing to operate; recovery often depends on most of the system continuing to operate
  - It reduces outage time by eliminating almost all of the time associated with an orderly shutdown (only to store some information about the event)
  - It simplifies the failure model by reducing the size of the recovery state table
  - It simplifies testing by reducing the failure combinations that need verification

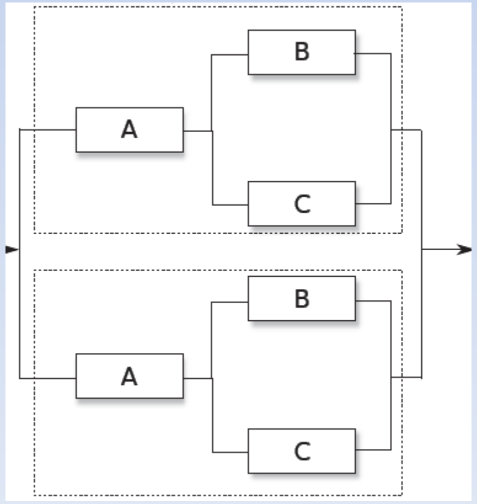
Replication and Diversification

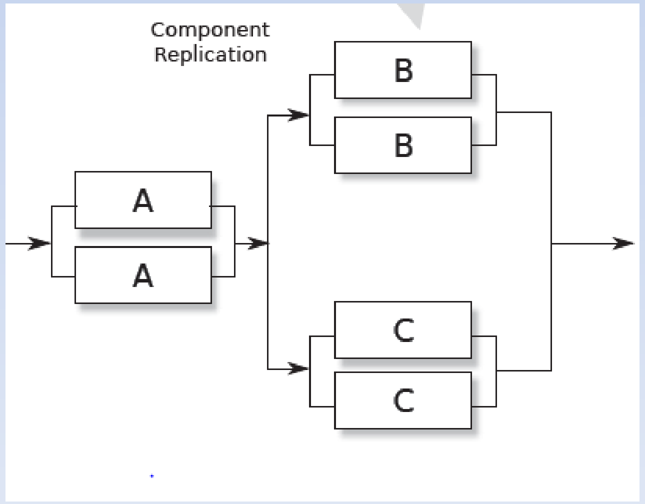
- The distinction between them is that replication deploys two or more identical copies of a system, whereas diversification incorporates different implementations of the same function
- The purpose of using replication or diversification is to increase either the availability or the reliability of a system
- Generally, requiring two systems to present the same response to a particular stimulus will increase reliability while decreasing availability (because both systems have to be running to provide any response)
- Replication and diversity are techniques recommended in both IEC 61508 and ISO 26262

Component or System Replication

- If replication or diversification is to be used, is it better to replicate components or systems: Which of the two designs in the lower part of the figure is likely to deliver higher dependability?







Component or System Replication

- The answer may vary from system to system, particularly if a system has several different failure modes, but in general, replicating components is better than replicating systems. This result follows from the inequality:

$$\phi(\vec{x} \coprod \vec{y}) \ \geq \ \phi(\vec{x}) \coprod \phi(\vec{y})$$

$$\coprod_{i=1}^{N} x_i = 1 - \{(1 - x_1) \times (1 - x_2) \times \ldots \times (1 - x_N)\}$$

- As an extra advantage, if the components are hardware, it may be unnecessary to replicate all of them to achieve the necessary dependability, leading to a product that is cheaper than would result from complete system replication.

The Structure Function

- A system that consists of N components can be described by a vector of N elements, each representing the state (1 = functioning, 0 = failed) of one component

$$\phi(\vec{x}) = \begin{cases} 0 \text{ if system has failed} \\ 1 \text{ if system is functioning} \end{cases}$$

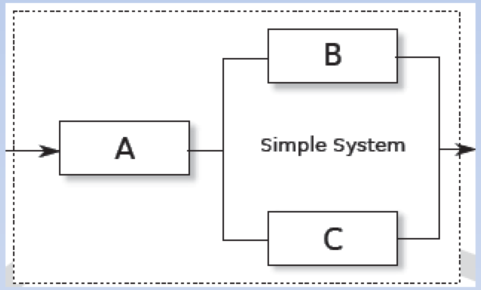
$$\vec{x} = (x_A, x_B, x_C)$$

$$\phi(1,1,1) = 1$$

$$\phi(1,1,0) = 1$$

$$\phi(1,0,0) = 0$$

$$\phi(0,1,1) = 0$$



Components in Parallel and Series

- Given this definition it is easy to see that for a set of components in series (as with old-fashioned Christmas tree lights), where all components must operate correctly for the system to operate

$$\phi(\vec{x}) = x_1 \times x_2 \times \ldots \times x_N = \prod_{i=1}^{N} x_i$$

- For components in parallel, where the system continues to operate until all components have failed:

- What is the value for a 2oo

$$\phi(\vec{x}) = 1 - (1 - x_1) \times (1 - x_2) \times \ldots \times (1 - x_N) = \coprod_{i=1}^{N} x_i$$

Components in Parallel and Series

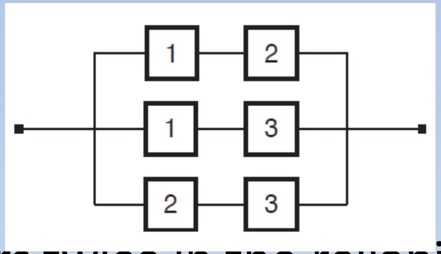
- A structure that is functioning if and only if at least k of the n components are functioning, is called a k-out-of-n structure
- The structure function of a k-out-of-n structure can be written

$$\phi(\mathbf{x}) = \begin{cases} 1 & \text{if} \quad \sum_{i=1}^{n} x_i \geq k \\ 0 & \text{if} \quad \sum_{i=1}^{n} x_i < k \end{cases}$$

- A k-out-of-n structure is often called a KooN structure. Notice that:
  - A 1oo1 structure is a single component
  - An NooN structure is a series structure of n components (All the n components must function for the structure to function)
  - A 1ooN structure is a parallel structure of n components (It is sufficient that one component is functioning for the structure to function)

2-out-of-3 structure

- A 2-out-of-3 (2oo3) structure is functioning when at least 2 of its 3 components are functioning



- Note that each component appears twice in the reliability block diagram
- Note also that since x_i is a binary variable, $x_k^i = x_i$ for all i and k.

$$\begin{aligned}\phi(\mathbf{x}) &= x_1x_2 \amalg x_1x_3 \amalg x_2x_3 \\ &= 1-(1-x_1x_2)(1-x_1x_3)(1-x_2x_3) \\ &= x_1x_2+x_1x_3+x_2x_3-x_1^2x_2x_3-x_1x_2^2x_3-x_1x_2x_3^2+x_1^2x_2^2x_3^2 \\ &= x_1x_2+x_1x_3+x_2x_3-2x_1x_2x_3\end{aligned}$$

2-out-of-3 structure

- Failure probability of a parallel system

$$P[\text{system failure}] = P[\ (Y_1 = 1) \cap (Y_2 = 1) \cap \dots \cap (Y_n = 1)\ ] = \prod^{n}_{i=1} P_i$$

- Failure probability of a serial system

$$P[\text{system failure}] = 1 - P[\text{system survival}] =$$

$$= 1 - P[\ (Y_1 = 0) \cap (Y_2 = 0) \cap \dots \cap (Y_n = 0)\ ] = 1 - \prod^{n}_{i=1} P_i$$

- In general, if the probabilities are the same, for K failures out of n

$$Pr(k; n, p) = \Pr(X = k) = \binom{n}{k} p^k (1-p)^{n-k}$$

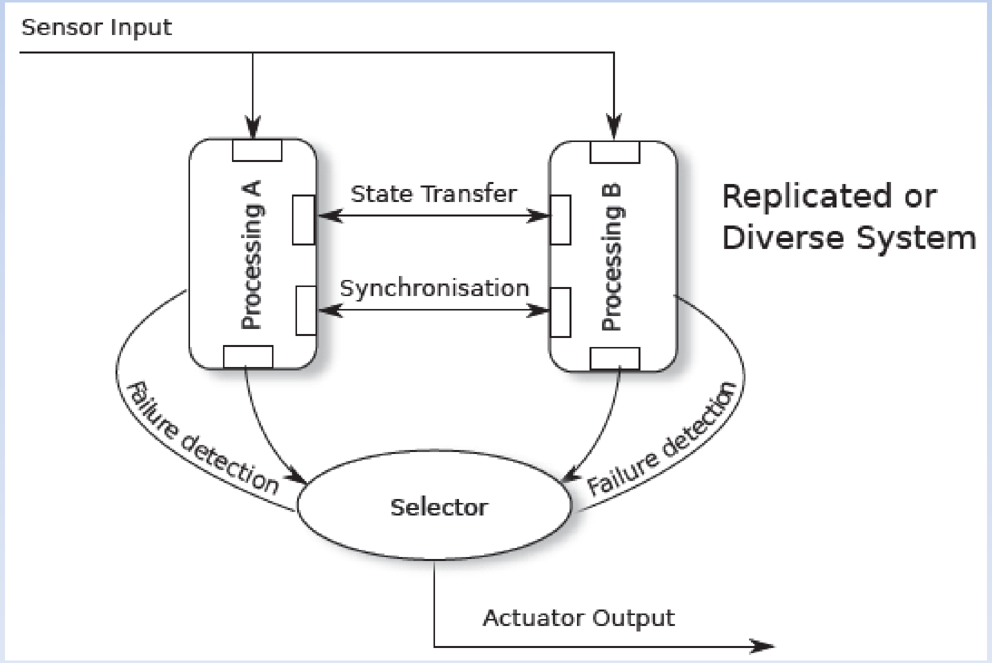
Examples

- **Example 1**. Consider a car with one spare tire. The car will become impaired if 2 (or more) tires are flat. In a conservative approximation, one may assume that all 5 tires are simultaneously subject to punctures.

- **Example 2**. In order to fly, an airplane needs at leat half of its engines to be functioning. Suppose that, during any given flight, engines fail independently, with probability P. Would you be safer in an airplane with 1, 2, 3 or 4 engines?

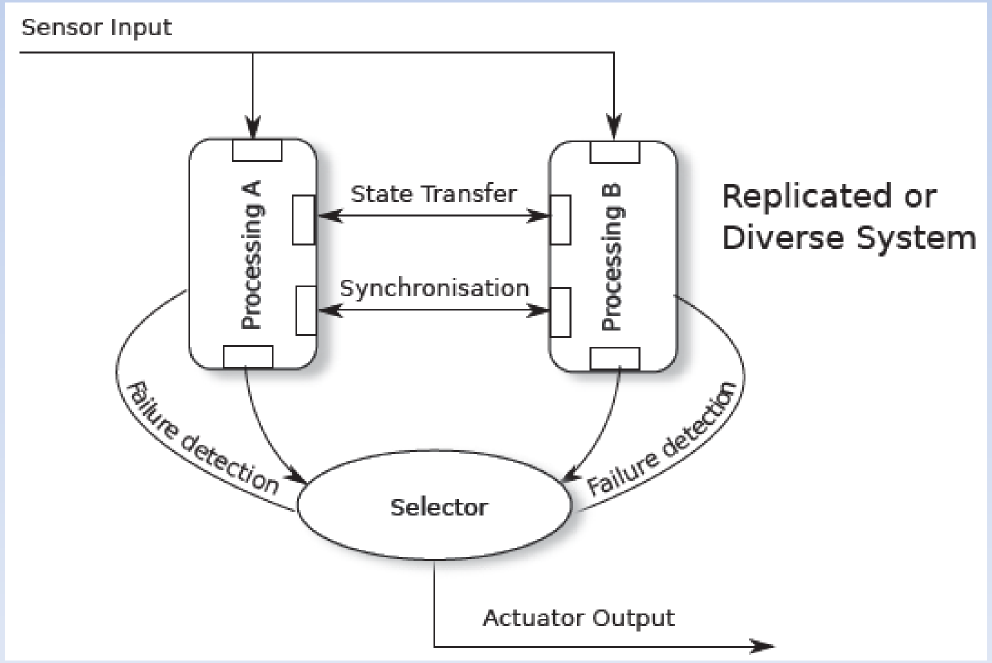
System Replication

- The drive for replication often stems from the view that, if a single system cannot meet the necessary dependability requirements, then by replicating it, a much higher level of dependability can be obtained
- This can be true if the replication is treated very carefully, but often the increased complexity of replication outweighs the advantages



System Replication

- The selector has been introduced to compare the two outputs and choose between them
  - Increase Reliability: compare the outputs (2oo2)
  - Increase Availability: pass the first intput (1oo2)
- **State transfer and synchronization**



Cold Stand By

- The “cold standby" model is a particular case where only one of the subsystems is operating and the other is simply monitoring its behavior, ready to take over if the active partner fails
- In this case the selector becomes a switch connecting the active partner to the actuators
- Some additional potential failure points are introduced:
  - How is a failure of the idle subsystem detected?
  - How does the selector (a switch) know that the active subsystem has failed and that a switch to the standby is required?
  - How does a failure of the selector itself affect the dependability of the overall system?
- Another disadvantage of the cold standby architecture is that it is difficult to scale. If the required level of dependability needs three subsystems, then the selector becomes very complex, and therefore error-prone

Time Replication

- Given that most bugs encountered in the field are Heisenbugs, simply running each computation twice and comparing the answers may be sufficient for the system to detect a bug.
- Performing the computation a third time and finding two identical answers may then be enough to obtain a reliable result, the adequacy of the design being assessed during the failure analysis
- This form of replication is known as “time replication," as the commodity being replicated is time: The system runs more slowly, but more reliably

Diversification: Hardware Diversity

- It would be possible for the two subsystems to be running on different processors. The benefit of this is unclear. Certainly, it would entail more work (different board support packages, different board layouts, different compilers, etc.), and we have to consider the types of processor fault against which we are trying to guard: Heisenbugs or Bohrbugs
- Heisenbugs can arise in processors: the errata for one popular processing chip contain statements from the chip manufacturer, such as:

  "Under very rare circumstances, Automatic Data prefetcher can lead to deadlock or data corruption. No fixed scheduled"

Diversification: Hardware Diversity

- Even without the chip errata, Heisenbugs can arise from electromagnetic interference, the impact of secondary particles from cosmic ray events or, for many memory chips, “row hammering”
- However, we do not need diversity to protect against Heisenbugs, we can use replication.
- If such a Heisenbug affects the operation of one processor, it is extremely unlikely to affect the other processor in the same way at the same time
- Diversification of hardware would be effective against Bohrbugs in the processor. However, these are very rare. The last reported Bohrbug might be the 1994 division bug in the x86 Pentium processor that could not calculate 4195835/3145727
- To accept the cost of a diverse processor design in case a chip manufacturer has introduced a Bohrbug may not be a cost-effective strategy

Software Diversity

- Code-level Diversity
  - the simplest is to use a tool to transform one source code into an equivalent one
- Coded Processors
  - The idea is an extension of the concept of program diversity, whereby a code is stored with each variable and is used to detect computational errors caused by incorrect compilers, hardware errors (bit flips)
  - As variables are manipulated (e.g., $z = x + y$ or $x > 0$), so are the appropriate codes. Software or hardware can then check the computation by checking the validity of the new codes, and incorrect task cycling
  - This technique is used in the driverless railway system in Lyon, on the Meteor line in Paris, and in the Canarsie subway line in New York

N-Version Programming

- Original definition:
  - N-version programming is defined as the independent generation of N >=2 functionally equivalent programs from the same initial specification. The N programs possess all the necessary attributes for concurrent execution, during which comparison vectors (“c-vectors") are generated by the programs at certain points. . . .
  - “Independent generation of programs" here means that the programming efforts are carried out by N individuals or groups that do not interact with respect to the programming process. Wherever possible, different algorithms and programming languages (or translators) are used in each effort

N-Version Programming

- Unfortunately, several studies have found that code faults are not independent in N-Version programming

For the particular problem that was programmed for this experiment, we conclude that the assumption of independence of errors that is fundamental to some analyses of N-Version programming does not hold. Using a probabilistic model based on independence, our results indicate that the model has to be rejected at the 99% confidence level

- This is probably because many faults are introduced due to ambiguity in the specifications that are common to all N teams. Moreover most faults appear in complex code and code that is complex for one team is likely to be complex for the others

Data Diversity

- Of all the forms of diversity, data diversity is perhaps the most useful, and least controversial
- The idea is to store the same information using two or more different data representations.
  - In an algorithm for calculating the braking of a train, for example, the same information might be held as data in both the time and frequency domains
- This provides a defense against both Bohrbugs and Heisenbugs in programs
- A program may, for example, include a Bohrbug that triggers whenever integer overflow occurs
- Such overflow might occur while processing the time series, but not when processing the frequency spectrum information.

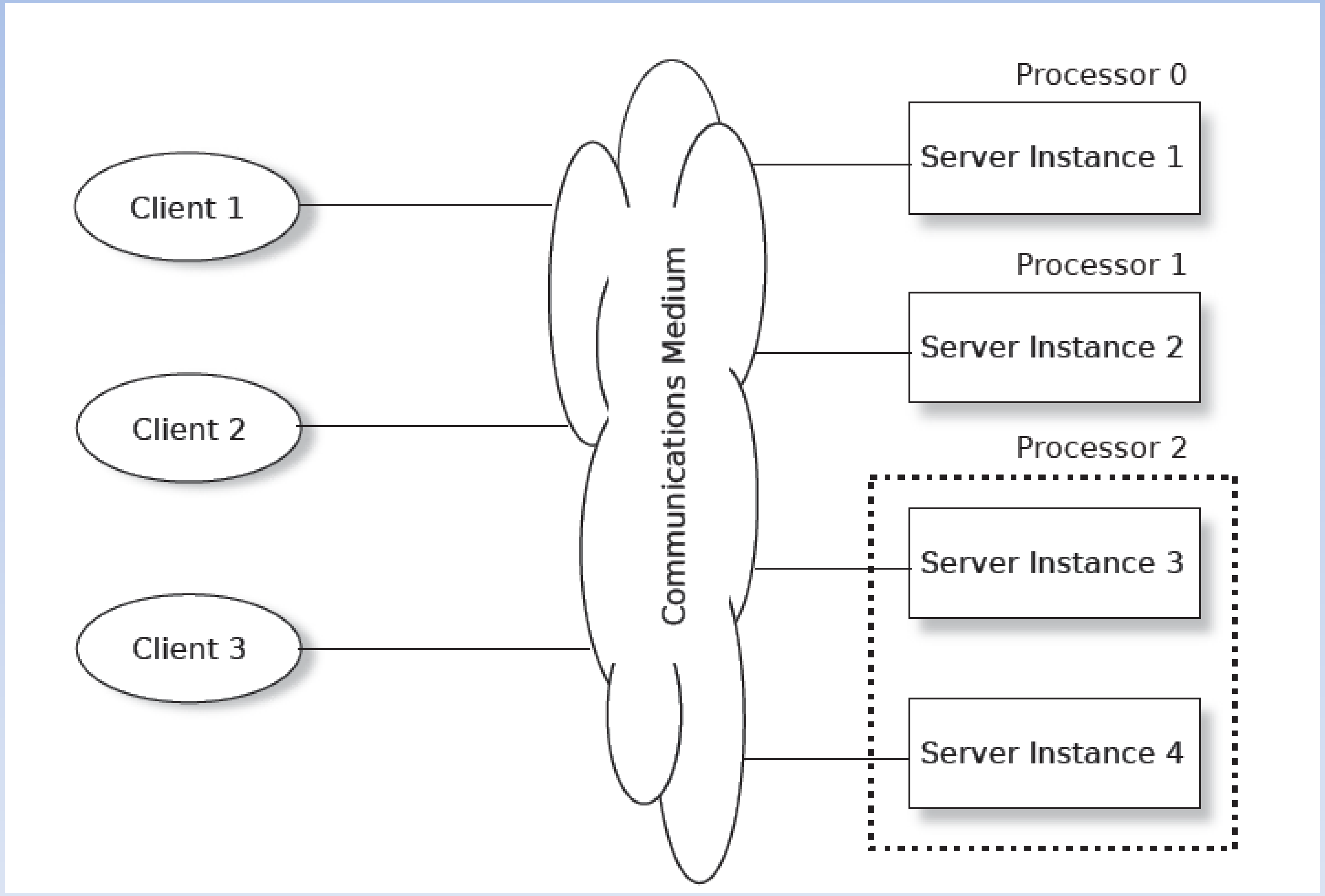
Active Replication

- Active replication, also called group synchrony, is a form of replication (or diversification) that can be very powerful in certain architectures:
  - Servers join groups and, when they join, they are provided with an up-to-date copy of the server group's state. Each server may, or may not, be aware of the other members of the group, but each clientmaking use of the service communicates with the group and believes that it is interfacing with a single server
  - As client messages arrive and group membership changes (servers joining or leaving), notification of these events is fed to all servers in the group in the same order.
  - So, if one member of the group sees that server X has left the group and then sees client Y's request, then all members of the group will see those two events in that order

Active Replication

- The underlying idea is that, if the servers all start in the same state (and the updating of a new group member's state is also part of the protocol) and if each server has received the same messages in exactly the same order, then all the servers must eventually arrive at the same state
- This is virtual synchrony, not locked-step operation
- For some systems, the requirement for common arrival order of all events at each server in the group can be relaxed and different types of guaranteed can be provided:
  - Sender order: All messages from each client are delivered to each group member in the order they were sent. There is no guarantee that messages from two different clients will be delivered in the same order to the group members
  - Causal order: If a message s1 causes a program to send message s2 (i.e., s1 causes s2), then s1 will be delivered before s2 to all group members
  - Total or agreed order: If any group member receives s1 before s2, then all group members will.

Active Replication



Active Replication

- The communication mechanism between the clients and the servers in the figure is indicated by a cloud.
- In principle, this can be any medium supporting messages (WAN, LAN a backplane in a rack,…)
  - The closer the connection, the better active replication works
- There are several technical solutions and implementations of this approach in the form of middleware of virtualization platforms
- Active replication is complex, and we deal with it in a separate presentation

But, What do we replicate?
Single big service performs poorly… why?

- Until 2005 “one server” was able to scale and keep up, like for Amazon’s shopping cart. A 2005 server often ran on a small cluster with, perhaps, 2-16 machines in the cluster.
- This worked well.
- But suddenly, as the cloud grew, this form of scaling broke. Companies threw unlimited money at the issue but critical services like databases still became hopelessly overloaded and crashed or fell far behind.

Jim Gray’s Famous Paper

- At Microsoft, Jim Gray anticipated this as early as 1996.

The dangers of replication and a solution. Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. 1996. In Proceedings of the 1996 ACM SIGMOD Conference. pp 173-182. DOI=http://dx.doi.org/10.1145/233269.233330

Basic message: divide and conquer is really the *only* option.

How their paper approached it

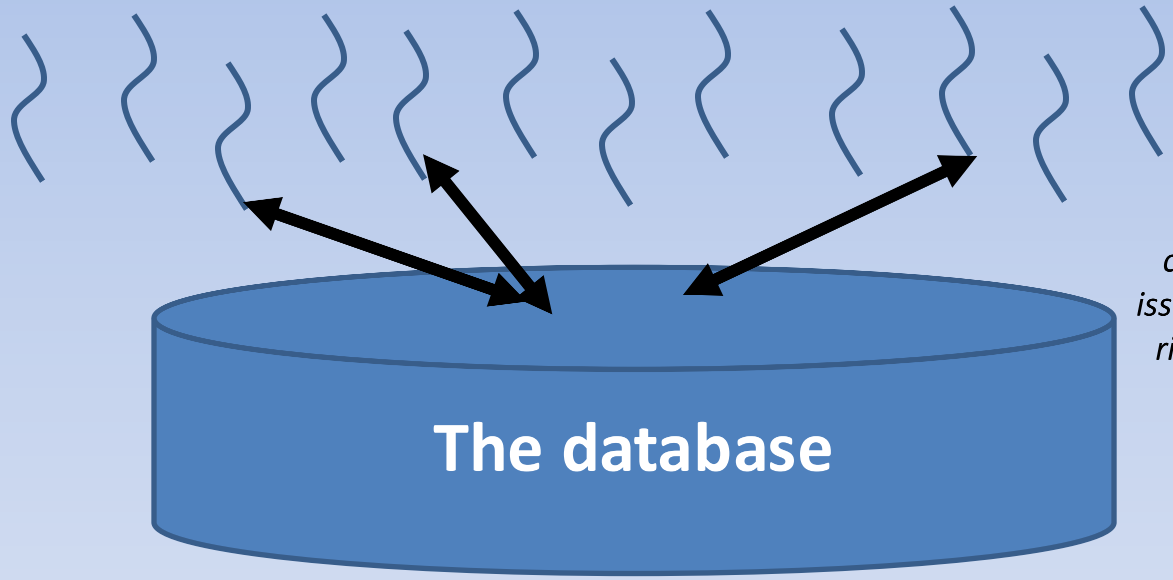
- The paper uses a “chalkboard analysis” to think about scaling for a system that behaves like a database.
    - It could be an actual database like SQL Server or Oracle
    - But their “model” also covered any other storage layer where you want strong guarantees of data consistency.
    - Mostly they talk about a single replicated storage instance, but look at structured versions too.

The core issue

- The paper assumes that your goal is some form of lock-based consistency, which they model as database serializability or "state machine replication", as in our case
- So, applications will be using read locks, and write locks, and because we want to accommodate more and more load by adding more servers, the work spreads over the pool of servers.
- This is a very common way to think about servers of all kinds.

Partially based on matrial from Ken Birman – Cornell

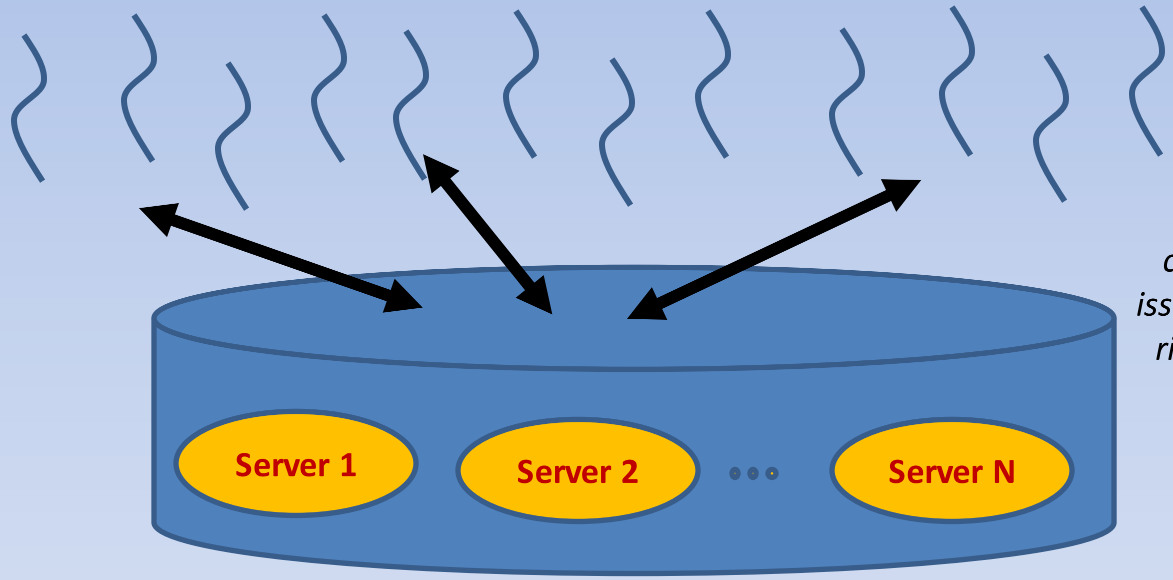
Their setup, as a picture



Applications using the database: *client processes*

During the run, T concurrent requests are issued. Here, 3 are running right now, but T could be much larger.

Their Basic setup, as a picture



Applications using the database: *client processes*

During the run, T concurrent requests are issued. Here, 3 are running right now, but T could be much larger.

For scalability, the number of servers (N) can be increased

What should be The goals?

- A scalable system needs to be able to handle “more T’s” by adding to N

- Instead, they found that the work the servers must do will increase as T^5

  Worse, with an even split of work, deadlocks occur as $N^{3,}$ causing feedback (because the reissued transactions get done more than once).

- Example: if 3 servers (N=3) could do 1000 TPS, with 5 servers the rate might drop to 300 TPS, purely because of deadlocks forcing abort/retry.

Why do services slow down at scale?

- The paper pointed to several main issues:
  - **Lock contention**. The more concurrent tasks, the more likely that they will try to access the same object (birthday paradox!) and wait for locks.
  - **Abort**. Many consistency mechanisms have some form of optimistic behavior built in. Now and then, they must back out and retry.
  - **Deadlock** also causes abort/retry sequences.

- The paper actually explores multiple options for structuring the data, but ends up with similar negative conclusions *except in one specific situation.*

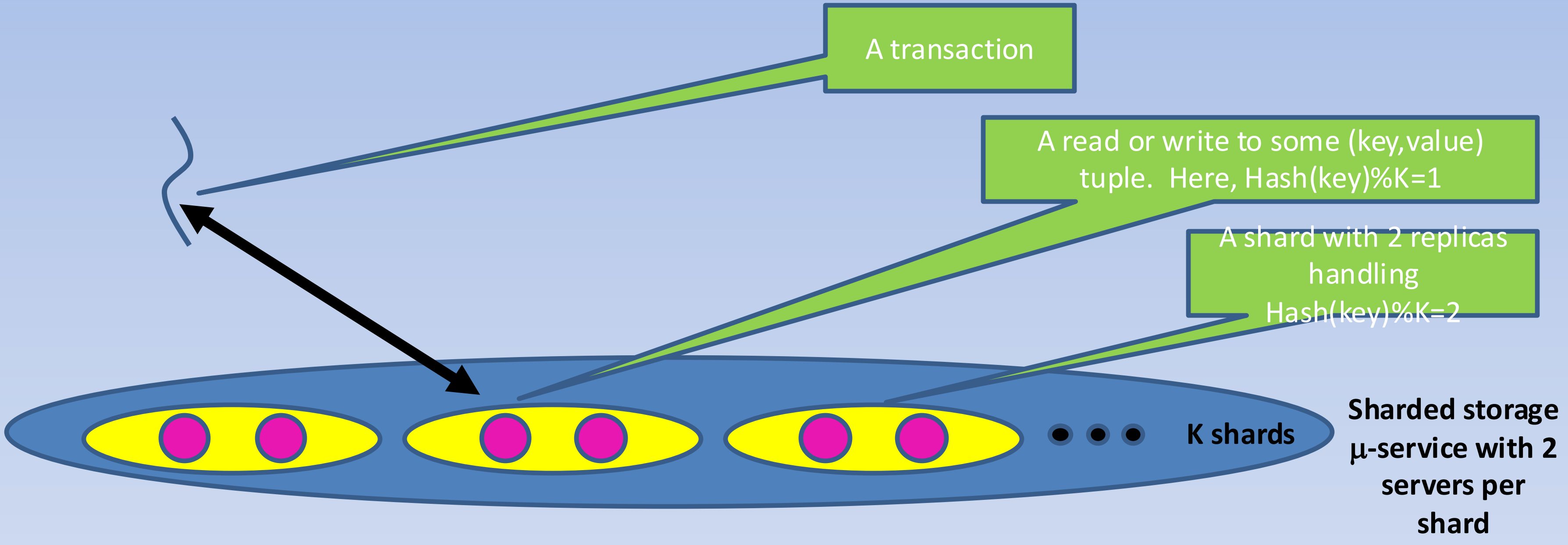
What Was that one good option?

- Back in 1996, they concluded that you are forced to shard the database into a large set of much smaller databases, with distinct data in each. Jim set out to do this for a massive database of astronomy data.

- By the time he died in 2007, Jim had shown that for every problem he ran into, it was possible to devise a sharded solution in which transactions only touched a single shard at a time.

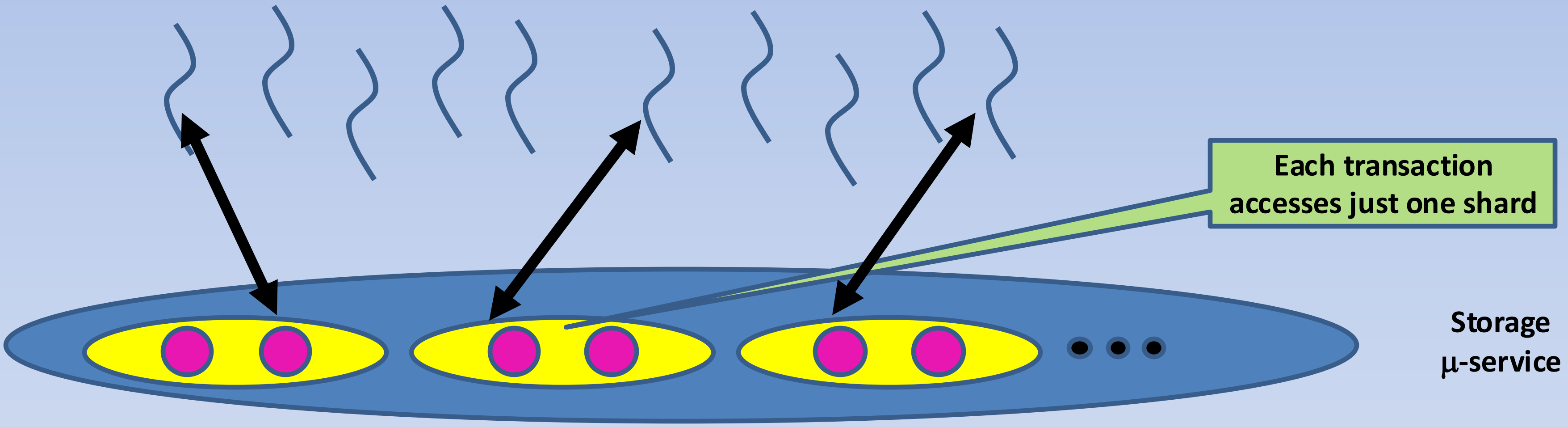
- In 1996, it wasn’t clear that every important service could be sharded. By the 2007 period, Jim had made the case that in fact, this is feasible!

Sharding: This works… but carefully


A transaction
A read or write to some (key,value) tuple. Here, Hash(key)%K=1
A shard with 2 replicas handling Hash(key)%K=2
K shards
Sharded storage μ-service with 2 servers per shard

- We will often see this kind of picture. Cloud IoT systems make very heavy use of key-based sharding. A (key,value) store holds data in the shards.

Sharding: This works… but carefully



- If a transaction does all its work at just one shard, never needing to access two or more, we can use *state machine replication* to do the work.
- No locks or 2-phase commit are required. This scales very well.

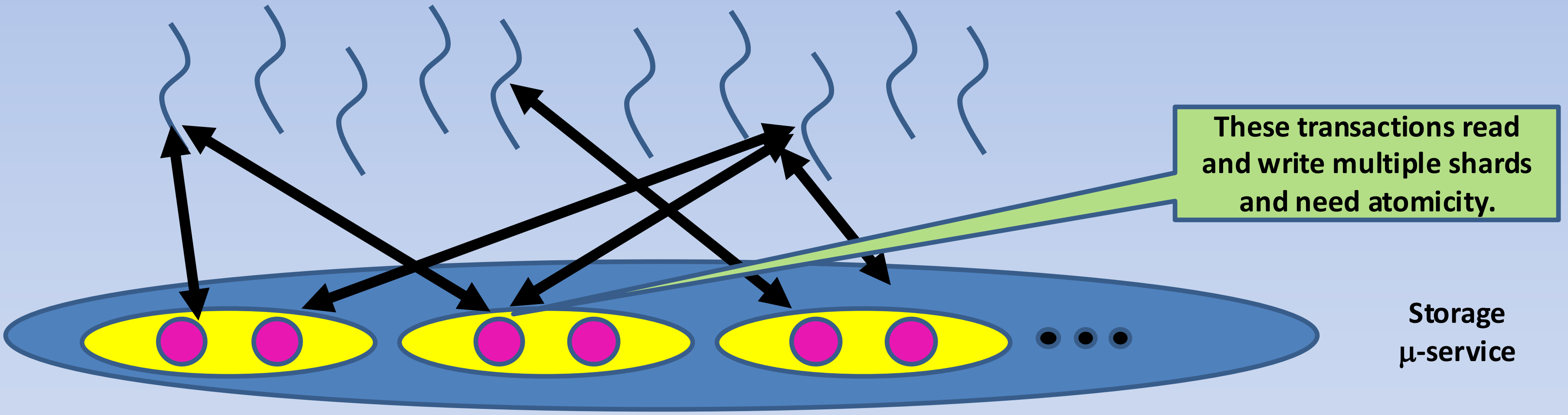
DefN: State Machine Replication

- This is a model in which we can read from any replica.

- To do an update, we use an *atomic multicast* or a *durable Paxos write* (multicast if the state is kept in-memory, and durable if on disk).

- The replicas see the same updates in the same sequence and so we can keep them in a consistent state. We package the transaction into a message so "delivery of the message" performs the transactional action.

Sharding fails for arbitrary transactions


These transactions read and write multiple shards and need atomicity.
Storage µ-service

- Transactions that touch multiple shards need locks and 2-phase commit.

  Jim Gray’s analysis applies: as we scale this case up, performance collapses.

Virtual Synchrony

- There are three clients in the example. Let us suposse the following scenario:
  - Client 1 and Client3 send in a request to what they believe is a single server, at more or less the same time
  - The virtual synchrony middleware, of which more later, replicates and orders the incoming requests and ensures that each of the four server instances receives the requests in the same order
  - Each server believes that it is alone and so performs whatever computation is required and replies with an answer
  - The virtual synchrony middleware accepts these answers, which arrive at different times, and takes a predefined action
    - In general, the action would be to send the first answer back to the client (or on to actuators) and discard the others
    - However, it could be configured to wait for a majority of server responses, compare them, and send the agreed value to the client

Active Replication

- Advantages over other replication schemes:
  - It is tolerant to Heisenbugs. It is unlikely that more than one instance have the same Heinsenbug at the same time. If a server crashes, it will disappear from the group, and a response from one of the other server instances will be returned to the client
  - It scales well (linearly) in the number of server instances. However, the time required to handle a transaction increases as N^3 as in other replication techniques
  - It can be tuned to provide reliability or availability
  - Silent failure (the main disadvantage of cold-standby) is avoided. All instances of the server are active all the time
  - The selector or switch is removed
  - State transfer is avoided. Except when a new member joins a group, there is no need to copy state from an “active" to a “passive" server as there is with a cold-standby architecture
  - Generally, no significant change is needed to either the client or the server to introduce virtual synchrony
  - It is a standardized technique

Active Replication

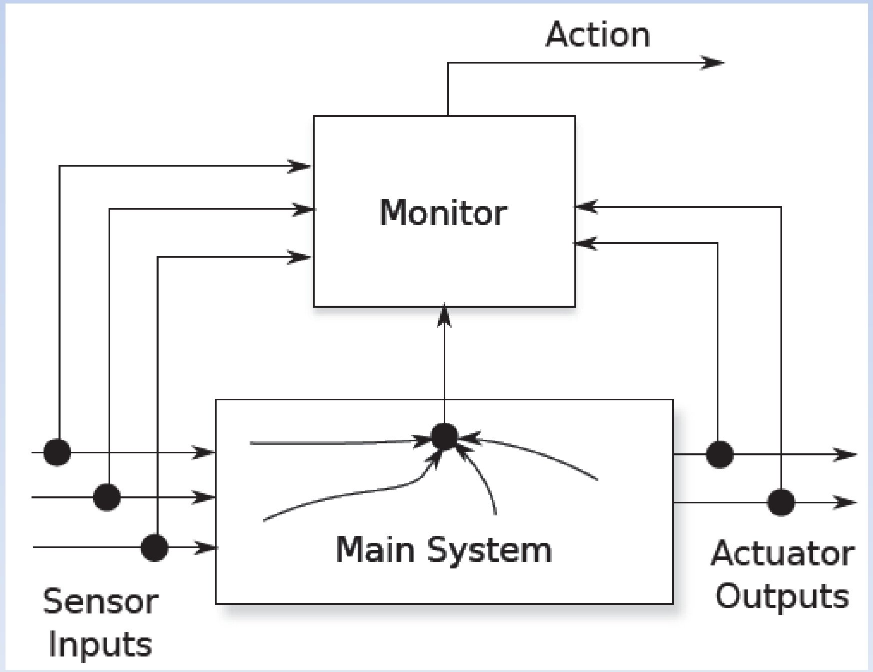
- Disadvantages:
  - All state changes in the server must take place as a result of received messages to ensure that the server states do not diverge. For example, a server instance may not generate a random number locally if that could alter its final state
  - It wastes processor time
  - The whole concept of group membership is technically very complex
    - **CAP theorem and FLP impossibility result**
    - If a server loses contact, with the other instances, it is impossible for those other instances to determine whether this is because the instance has crashed or because it is still running, communications have been broken
    - There are pragmatic ways of avoiding the problem

Locked step processors

- One surprisingly popular form of replication has two processors running the same code in locked step, each executing the same machine code instruction at the same time
- Initially (early 80's), software was very simple, and processor hardware very complex and error prone
  - the locked step provided a useful way of detecting a hardware fault
- Hardware became much more integrated and reliable and the technique became useless because almost all faults were in software
- In the last few years, the track widths in the integrated circuits have reduced. In 2005, track widths of 100 nm were typical; by 2010, this had reduced to 32 nm, and by 2018, tracks of 8 nm are starting to be used
  - Processors and memory chips have become more susceptible to cross-talk between tracks, thermal aging, electromagnetic interference, and cosmic ray effects
  - Locked step processing may again become useful

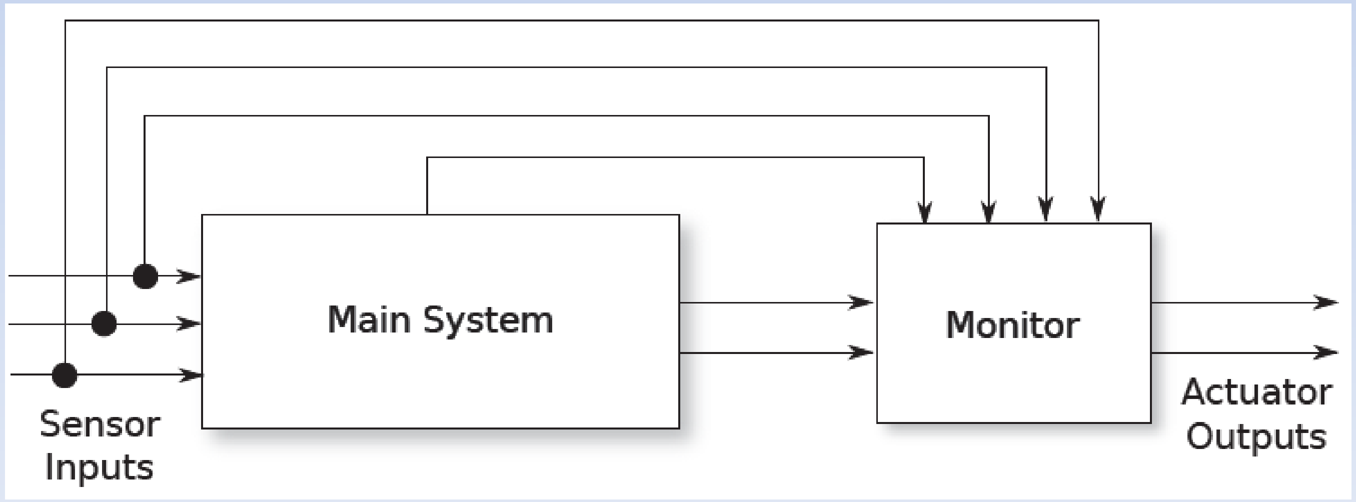
Diverse Monitor

- The main system performs its function of reading input values and transforming these into output values
- The monitor observes the inputs and outputs, and may also observe internal states of the main system
- The difference between the main system and the monitor is that the main system is much more complex



Diverse Monitor

- When designing a high reliability system, for example an ASIL C, sometime is permitted to decompose a system into an ASIL C component and a “quality management" (QM) component to which ISO 26262 does not apply
- The main system could be governed by the QM processes and the monitor (easier to certify) by the ASIL C
- There are other diverse monitor architectures, in the figure a drug flow medical device



Watchdog Diverse Monitor

- One very common diverse monitor is the watchdog
- The main system is required to “kick“ the watchdog (the monitor) periodically by sending it a message or by changing a memory value
- If the watchdog notices that it hasn't been kicked for a pre-determined time, then it “barks“, typically forcing the system into its design safe state
- The difficulties with the watchdog approach are determining:
  - The place within the main system's code to kick the watchdog
    - The process kicking the watchdog must be sophisticated enough to be able to determine whether the system is working correctly or not
  - The time within which the watchdog must be kicked

Replication and Diversification Summary

- Replication and diversification are powerful techniques for improving the availability or reliability of a system
- However, their use is not free: they add extra components and complexity to the system, and an unthinking application of replication may actually result in a decrease in dependability
- Silent failure of a component that is in cold standby mode is particularly pernicious

Diseño de Software Crítico

- Patrones para gestión de errores
  - Detección de anomalías: cadenas de Markov y filtros de Kalman
  - Manejo de errores: Rejuvenation y Bloques de Recuperación
- Replicación y diversificación
- Balances arquitectónicos: disponibilidad/fiabilidad, seguridad/prestaciones,…

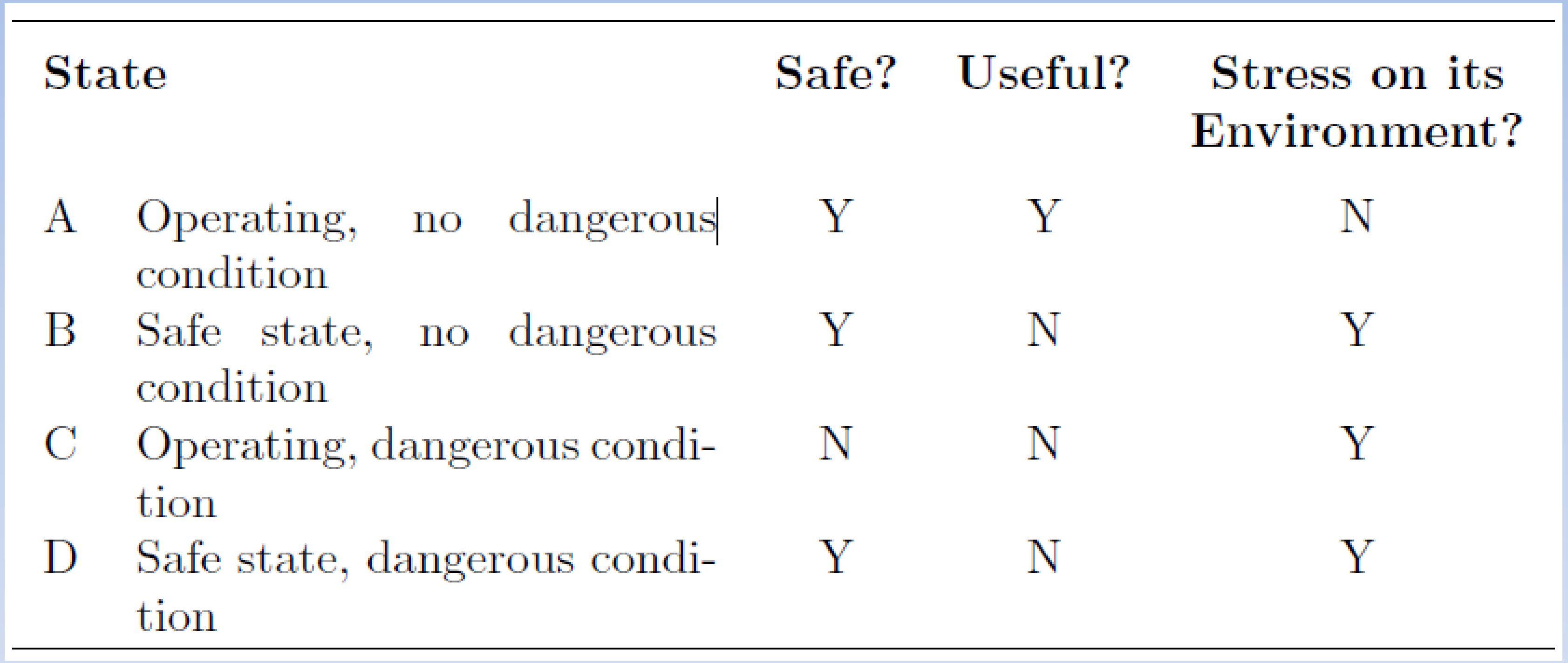
Architectural Balancing

- In this chapter we have reviewed a number of architectural and design patterns:
  - Patterns for error detection, including anomaly detection
  - Patterns for handling errors when they occur
  - Patterns for replication and diversification
- Selecting from amongst these patterns means acknowledging that many architectural balances have to be achieved

Usefulness/Safety Balance

- It is trivially easy to build a very safe system: a train that never moves, traffic lights that are permanently red,…
- The balance that has to be achieved is that between usefulness and safety
- Sometimes a highly reliable system (e.g., 2oo2) can easily degrade
- A device moving to its design safe state generally becomes useless, but worse, it puts stress onto the larger system and the environment

Usefulness/Safety Balance



State		Safe?	Useful?	Stress on its Environment?
A	Operating, no dangerous condition	Y	Y	N
B	Safe state, no dangerous condition	Y	N	Y
C	Operating, dangerous condition	N	N	Y
D	Safe state, dangerous condition	Y	N	Y

Security/Performance/Safety

- **Usability** is about making systems easy to use for everyone
- **Security** is about making systems easy to use for designated people and very hard (if not impossible) for every one else
- **Safety** is about stopping the right people doing bad things. Good people make slips and errors; in a safe system those errors should not escalate to untoward incidents, harm or other safety issues.

Security/Performance/Safety

"A secure system may not be very usable. In turn this will encourage its users to find workarounds to make their daily use of the system more `user friendly'.

Thus a hospital system may end up with everyone sharing the same password, as this is much easier than remembering individual passwords and logging in and out repeatedly"

Implementation Balance

- As the programmers implement the system, they will need guidance on how to program
- It is possible to program for high performance (fast, tightly-knit code), for testability, for ease of maintenance, for efficiency of the static analysis tools, for efficient runtime error detection
- These demands work against each other:
  - High-performance code is likely to be less readable, less easily maintained, less likely to detect runtime errors, and less amenable to deep static analysis
  - Programmers need to be aware of the priorities for each module
  - Once these priorities have been set for each module, the resulting characteristics can be built into the system's failure model