# Task 1.1: Basic Problems

1. In einem verteilten System haben die unterschiedlichen Systeme/Einheiten naturgemäß keinen Zugriff auf den Zustand des Gesamtsystems (zumindest nicht, ohne unannehmbare Zeiteinbußen in Kauf zu nehmen). Das heißt, es gibt keine „globalen“ Variablen bzw. keinen geteilten Speicher. Das erschwert die Synchronisierung in einem verteilten System und kann dazu führen, dass Berechnungen ggf. auf bereits veralteten Daten ausgeführt werden.
2. Die Zeit auf den verschiedenen Systemen ist unter Umständen nicht 100% synchronisiert. Dies führt dazu, dass es nicht mehr möglich ist für ein System exakt zu bestimmen welche Nachrichten von welchen Systemen in welcher Reihenfolge gesendet wurden, da der hinterlegte Zeitstempel sich auf die lokale Zeit der jeweiligen Systeme bezieht.
3. Systeme erzeugen unter Umständen indeterministische Ausgaben. Das heißt, die Ausführung der selben Aufgabe kann (z.B. aufgrund von race conditions) zu unterschiedlichen Ausgaben führen. Zudem können durch die Übertragung von Daten über das Netzwerk (mit einem entsprechenden Protokoll) weitere Fehler entstehen.

# Task 1.2: Abstraction Levels

**Level 1: Physical Configuration**  
Beschreibt den Aufbau des physikalischen Netzwerkes über welches die verteilten Systeme kommunizieren. Das ist wichtig, da die Bandbreite, Latenz, Zuverlässigkeit und ggf. Kosten bei der Übertragung wichtige Kennziffern sind welche bei der Implementierung von verteilten Systemen berücksichtigt werden müssen.

**Level 2: Logical Configuration**  
Beschreibt den Aufbau des Netzwerkes auf abstraktere Weise. Primär werden hier die „communication subsystems“ (CSS) betrachtet. Diese beschreiben die Vernetzung von „ähnlichen“ Systemen (=Klassen von Systemen) innerhalb des Netzwerkes. Dabei kann ein System Teil von mehreren CSS sein.

**Level 3: Logical Distribution (Process network)**  
Hier beginnt die Abstrahierung der „Entitäten“ eines verteilten Systems. Eine Entität beschreibt eine Instanz in dem Netzwerk welche Nachrichten von anderen Entitäten empfangen kann bzw. selbst Nachrichten an andere Entitäten verschicken kann. Eine Entität muss dabei kein physikalisches System sein, sondern kann z.B. einen lokalen Prozess repräsentieren, welcher mit anderen Prozessen kommunizieren kann (IPC).

**Level 4: Distributed Algorithm**  
Die letzte Stufe abstrahiert letztendlich auch die Umgebung des verteilten Systems, d.h. dessen Performanz und Zuverlässigkeit im Allgemeinen bzw. der Umgebung der einzelnen Systeme.

# Task 1.3: Transparency

Der Begriff beschreibt letztendlich die Abstrahierung des verteilten Systems gegenüber dem Anwender/Programmierer der mit dem System arbeitet. Ziel ist es, dass das System als ein „Ganzes“ wahrgenommen wird, und nicht als eine Menge von unterschiedlichen Knoten/Systemen.

Beispiele:

1. **Access transparency**  
   Der Zugriff auf eine logische Ressource erfolgt immer gleich – unabhängig davon ob es sich um eine Ressource handelt die sich physikalisch auf dem gleichen System oder auf einem anderen System befindet (z.B. Network File System (NFS))
2. **Location transparency**  
   Die Kenntnis des physikalischen Standortes eines System (z.B. in einem konkreten Rack eines Rechenzentrums) ist nicht relevant um mit dem System kommunizieren zu können. Stattdessen reicht auch hier eine logische Adressierung (z.B. in Form eines Hostnames (DNS))
3. **Concurency transparency**Mehrere Prozesse können nebenläufig auf denselben Ressourcen arbeiten, ohne sich dabei gegenseitig in die Quere zu kommen (z.B. durch einen LOCK bei einem schreibenden Zugriff auf eine Datenbank / eine Datei).
4. **Replication transperency**Eine Ressource kann physikalisch redundant im verteilten System vorliegen, beispielsweise um die Performanz bei einem Lesezugriff zu erhöhen indem Teile der Datei von mehreren Systeme angefordert werden (z.B. RAID, HDFS)

# Task 1.4: Programming Abstractions

1. **Distributed Operating System Approach**  
   Das Konzept der verteilten Programmierung ist hier ein fester Bestandteil des Betriebssystems. Somit ist es (nahezu) Sprachenunabhängig - d.h. es funktioniert mit C++ genauso wie mit jeder anderen Programmiersprache die auf die Betriebssystemfunktionen zurückgreifen kann. Jedoch ist keine Plattformunabhängigkeit gegeben, da die Programme an das entsprechende Betriebssystem gebunden wären.
2. **Distributed Database Approach**  
   Bei dem Ansatz einer verteilten Datenbank ist zwar die Platform- und Sprachenunabhängigkeit gegeben, jedoch lassen sich viele verteilte Algorithmen mit diesem Ansatz nur schwer realisieren.
3. **Protocol Approach (for dedicated purposes)**  
   Die Nutzung von Protokollen zur Kommunikation zwischen verteilten Systemen erlaubt komplette Sprach-/ und Plattformunabhängigkeit bei einer guten Performanz. Jedoch sind erfüllen Protokolle nur sehr spezifische Aufgaben (bzw. meistens exakt eine einzige Aufgabe – diese aber dafür sehr gut).
4. **Distributed Programming Language Approach**

Bei diesem Ansatz wird Sprachenspezifische Middleware (meist in Form von eigenen Bibliotheken) verwendet um die verteilte Programmierung über ein Netzwerk von mehreren Systemen zu erlauben.