# Nachrichten / Kommunikation zw. Server und Client

1. Grundsätzlich sendet der Server jedes Ereignis inkl. Private- und PublicState (oder nur die Status-Änderungen, was ist besser?) über alle aktiven Websocket-Verbindungen, damit die menschlichen Mitspielern auf dem aktuellen Stand bleiben können. Das ist für Agenten nicht notwendig, da sie bei einer Entscheidung direkt auf PublicState und PrivateState zugreifen können.
2. Sobald der Client sich verbindet, sendet er den gewünschten Tisch, an den er sich setzen möchte.
3. Wenn der Client das Spiel regulär verlassen will, kündigt der Client dies an, damit der Server nicht erst noch 20 Sekunden wartet, bis de Client durch eine KI ersetzt wird.
4. Der erste Client (Mensch) am Tisch darf die Sitzplätze der Mitspieler bestimmen, bevor er das Spiel startet (normalerweise wird er warten, bis sein Freunde auch am Tisch sitzen und dann sagen, wer mit wem ein Team bildet.)
5. Eine Neue Runde beginnt. Der Server verteilt je 8 Karten an jeden Spieler.
6. Jeder Spieler muss sich dann entscheiden, ob er Grand Tichu ansagen möchte oder nicht (passt).
7. Sobald jeder Client sich entschieden hat, ob er ein Großes Tichu ansagen möchte oder nicht, teilt er die restlichen Karten aus (je 6 pro Spieler).
8. Solange noch kein Spieler Karten zum Tausch (Schupfen) abgegeben hat, kann der Spieler ein Tichu ansagen.
9. Die Spieler müssen nun 3 Karten zum Tauschen abgeben (verdeckt, je eine pro Mitspieler).
10. Sobald alle Spieler die Karten zum Tauschen abgegeben haben, sendet der Server an jedem Client jeweils die getauschten Karten, die für dem Client bestimmt sind.
11. Ab jetzt kann der Spieler so wie vor dem Schupfen wieder Tichu ansagen, solange er noch 14 Karten auf der Hand hat.
12. Der Spieler mit dem MahJong muss eine Kartenkombination ablegen.
13. Der nächste Spieler wird aufgefordert, Karten abzulegen oder zu Passen.
14. Punkt 13 wird wiederholt, bis alle Mitspieler hintereinander gepasst haben, so das der Spieler, die die letzten Karten gespielt hat, wieder an der Reihe ist.
15. Dieser Spieler darf die Karten kassieren.
16. Wenn ein Spieler keine Handkarten mehr hat, kann er in die Karten der Mitspieler kucken (die Karten sind für ihn nicht mehr verdeckt).
17. Wenn die Runde beendet ist, und die Partie noch nicht entschieden ist (kein Team hat 1000 Punkte erreicht), leitet der Server automatisch eine neue Runde ein (wir beginnen wieder bei Punkt 5).
18. Wenn die Partie beendet ist, beginnen wir wieder mit Punkt 4.

**Diskussion zu deinem Entwurf und Nachrichten-Definition:**

1. **Punkt 1 (State Updates):**
   * **Diskussion:** Deinen Vorschlag, *jedes* Ereignis + *kompletten* State an *alle* zu senden, sollten wir **anpassen**. Das ist sehr ineffizient und sendet private Daten an die Falschen.
   * **Besserer Ansatz:**
     + **Gezielte State Updates:** Sende den PublicState nur an die Spieler *am betreffenden Tisch*, wenn sich etwas daran ändert. Sende den PrivateState *nur* an den jeweiligen Spieler, wenn sich dessen Hand oder privater Status ändert.
     + **Event-Nachrichten:** Sende zusätzlich spezifische Event-Nachrichten für wichtige Ereignisse (z.B. "Karte gespielt", "Tichu angesagt", "Stich gewonnen"). Das macht die Client-Logik einfacher, da sie auf konkrete Ereignisse reagieren kann, anstatt immer den ganzen State zu vergleichen.
     + **Delta-Updates (Optional/Zukunft):** Später könnte man statt des vollen States nur die Änderungen senden, das ist aber komplexer zu implementieren. Für den Anfang sind volle (aber gezielte) State-Updates einfacher.
     + **Agenten:** Agenten brauchen keine WebSocket-Nachrichten, sie interagieren direkt mit den State-Objekten in der Engine.
2. **Punkt 2 (Verbinden/Tisch wählen):**
   * **Kommunikation:** Das ist bereits durch die URL-Parameter (ws://...?tableName=X&playerName=Y&playerId=Z) und die Logik in GameFactory.handle\_connection\_request abgedeckt. Der Server antwortet mit joined\_table oder schließt die Verbindung bei Fehler.
   * **Nachrichten:**
     + C -> S: (Implicit via WebSocket URL Params)
     + S -> C: {"type": "joined\_table", "payload": {"tableName": ..., "playerName": ..., "playerId": ..., "playerIndex": ...}} (Bei Erfolg)
     + S -> C: (WebSocket Close mit Fehlercode/Nachricht bei Fehler)
3. **Punkt 3 (Reguläres Verlassen):**
   * **Diskussion:** Gute Idee, um den Timeout zu umgehen. Der Server kann den Spieler dann sofort entfernen oder durch eine KI ersetzen.
   * **Nachrichten:**
     + C -> S: {"action": "leave\_table", "payload": {}}
     + S -> C: (Server bestätigt ggf. mit player\_left Event oder schließt die Verbindung)
4. **Punkt 4 (Setup/Spielstart durch Spieler):**
   * **Client verbindet sich:** (Wie gehabt, mit Tischname) -> joined\_table. Der Tisch ist jetzt im "Lobby"-Modus.
   * **Spieler am Tisch:** Der PublicState muss anzeigen, wer an welchem Slot sitzt (Spielername oder "Leer" oder "KI"). Der Server sendet public\_state\_update nach jedem Beitritt/Verlassen.
   * **Lobby-Phase (Spieler hinzufügen/entfernen/KI setzen; Host-Aktion):** Ein Spieler (z.B. der erste am Tisch, der "Host") braucht Aktionen, um das Setup zu ändern.
     + C -> S: {"action": "assign\_slot", "payload": {"slot\_index": 2, "player\_type": "ai", "ai\_level": 1}}
     + C -> S: {"action": "assign\_slot", "payload": {"slot\_index": 1, "player\_type": "human", "player\_id": "UUID\_des\_freundes\_am\_tisch"}} (um Plätze zu tauschen)
     + C -> S: {"action": "assign\_slot", "payload": {"slot\_index": 3, "player\_type": "empty"}} (Slot leeren)
     + S -> C: public\_state\_update (Broadcast nach jeder Änderung im Lobby)
   * **Spiel starten (Host-Aktion):** Wenn das Setup passt (4 Spieler zugewiesen), kann der Host das Spiel starten.
     + C -> S: {"action": "start\_game", "payload": {}} (Host only)
     + S -> C: (Server validiert, startet intern die Engine und sendet dann die Nachrichten für Rundenstart - Punkt 5)
     + S -> C: {"type": "error", "payload": {"message": "Spiel kann nicht gestartet werden (z.B. nicht 4 Spieler)."}} (Unicast an Host bei Fehler)
   * **Wer ist Host?** Einfachste Regel: Der Spieler an Index 0 ist der Host und darf Setup-Änderungen und den Start auslösen. Wenn Spieler 0 geht, wird der nächste Mensch zum Host (z.B. Index 1). Wenn keine Menschen mehr da sind, wird der Tisch aufgeräumt (wie gehabt).
5. **Punkt 5 (Neue Runde / 8 Karten austeilen):**
   * **Kommunikation:** Server mischt, teilt 8 Karten aus, aktualisiert die PrivateState-Objekte.
   * **Nachrichten:**
     + S -> C: {"type": "event", "payload": {"event\_name": "round\_start"}} (Optional, Broadcast an Tisch)
     + S -> C: {"type": "private\_state\_update", "payload": {...PrivateState mit 8 Karten...}} (Unicast an jeden Spieler)
     + S -> C: {"type": "public\_state\_update", "payload": {...PublicState...}} (Optional, falls sich was Öffentliches ändert, z.B. Rundenzähler)
6. **Punkt 6 (Grand Tichu ansagen):**
   * **Kommunikation:** Server muss auf Entscheidung von menschlichen Spielern warten.
   * **Nachrichten:**
     + S -> C: {"type": "request\_action", "payload": {"request": "announce\_grand\_tichu", "timeout": 15}} (Unicast an jeden menschlichen Spieler, mit Timeout?)
     + C -> S: {"action": "announce\_grand\_tichu", "payload": {"announce": true/false}}
     + S -> C: {"type": "event", "payload": {"event\_name": "player\_announced\_grand\_tichu", "player\_index": X, "announced": true/false}} (Broadcast an Tisch)
7. **Punkt 7 (Restliche 6 Karten austeilen):**
   * **Kommunikation:** Server teilt aus, aktualisiert PrivateState.
   * **Nachrichten:**
     + S -> C: {"type": "private\_state\_update", "payload": {...PrivateState mit 14 Karten...}} (Unicast an jeden Spieler)
     + S -> C: {"type": "event", "payload": {"event\_name": "dealing\_complete"}} (Optional, Broadcast)
8. **Punkt 8 & 11 (Kleines Tichu ansagen):**
   * **Kommunikation:** Spieler kann jederzeit *vor seinem ersten gespielten Zug* Tichu ansagen. Server validiert.
   * **Nachrichten:**
     + C -> S: {"action": "announce\_tichu", "payload": {"announce": true}} (Spieler sendet dies proaktiv, wenn er will)
     + S -> C: {"type": "event", "payload": {"event\_name": "player\_announced\_tichu", "player\_index": X, "announced": true/false}} (Broadcast an Tisch nach Validierung durch Server)
     + S -> C: {"type": "error", "payload": {"message": "Tichu kann jetzt nicht angesagt werden."}} (Unicast, falls ungültig)
9. **Punkt 9 (Karten zum Schupfen wählen):**
   * **Kommunikation:** Server fordert Karten an und wartet auf alle Menschen.
   * **Nachrichten:**
     + S -> C: {"type": "request\_action", "payload": {"request": "select\_schupf\_cards", "timeout": 30}} (Unicast an jeden menschlichen Spieler)
     + C -> S: {"action": "submit\_schupf\_cards", "payload": {"to\_left": "RK", "to\_partner": "G5", "to\_right": "S2"}}
10. **Punkt 10 (Getupfte Karten erhalten):**
    * **Kommunikation:** Server führt Tausch durch, aktualisiert PrivateState.
    * **Nachrichten:**
      + S -> C: {"type": "private\_state\_update", "payload": {...PrivateState mit neuer Hand...}} (Unicast an jeden Spieler)
      + S -> C: {"type": "event", "payload": {"event\_name": "schupfen\_complete"}} (Broadcast an Tisch)
11. **Punkt 12 (Spielbeginn / Mah Jong):**
    * **Kommunikation:** Server bestimmt Startspieler (mit Mah Jong), fordert zur Aktion auf. Mah Jong muss als erstes gespielt werden. Wenn Mah Jong eine Zahl ist, muss evtl. ein Wunsch abgefragt werden.
    * **Nachrichten:**
      + S -> C: {"type": "your\_turn", "payload": {"player\_index": X, "must\_play\_mah\_jong": true, "possible\_actions": [...]}} (Unicast an Startspieler)
12. **Punkt 13 (Normaler Zug):**
    * **Kommunikation:** Server fordert nächsten Spieler auf.
    * **Nachrichten:**
      + S -> C: {"type": "your\_turn", "payload": {"player\_index": X, "possible\_actions": [...]}} (Unicast an aktuellen Spieler. possible\_actions enthält Infos über gültige Züge oder Passen)
      + C -> S: {"action": "play\_cards", "payload": {"cards": ["B7", "B7"]}}
      + C -> S: {"action": "pass\_turn", "payload": {}}
      + S -> C: {"type": "event", "payload": {"event\_name": "player\_played\_cards", "player\_index": X, "cards": [...], "combination\_info": {...}}} (Broadcast nach gültigem Zug)
      + S -> C: {"type": "event", "payload": {"event\_name": "player\_passed", "player\_index": X}} (Broadcast nach Passen)
      + S -> C: {"type": "public\_state\_update", "payload": {...PublicState...}} (Broadcast, um last\_played\_combination und current\_turn\_index zu aktualisieren)
13. **Punkt 14 & 15 (Stich Ende / Karten kassieren):**
    * **Kommunikation:** Server erkennt Stich-Ende, bestimmt Gewinner, aktualisiert Punkte (im State).
    * **Nachrichten:**
      + S -> C: {"type": "event", "payload": {"event\_name": "stich\_won", "winner\_index": X, "stich\_points": Y, "stich\_cards": [...]}} (Broadcast)
      + S -> C: (your\_turn-Nachricht an den Gewinner des Stichs zum Eröffnen des nächsten Stichs)
14. **Punkt 16 (Spieler leer / In Karten gucken):**
    * Wenn ein Spieler leer wird, muss der Server dies erkennen.
    * Der Server sendet dann eine spezielle Nachricht *nur an diesen leeren Spieler*, die die Hände der anderen enthält.
    * **Nachrichten:**
      + S -> C: {"type": "all\_hands\_update", "payload": {"hands": [{"player\_index": 1, "cards": ["Hu", "DR", ...]}, {"player\_index": 2, "cards": [...]}, {"player\_index": 3, "cards": [...]}]}} (Unicast an den leeren Spieler. hands enthält Infos für alle *anderen* Spieler).
    * **Wichtig:** Normale private\_state\_update-Nachrichten werden an diesen leeren Spieler nicht mehr gesendet (er hat ja keine Hand mehr). public\_state\_update und event-Nachrichten erhält er weiterhin.
15. **Punkt 17 (Rundenende / Nächste Runde):**
    * **Kommunikation:** Server erkennt Rundenende (ein Team oder 3 Spieler leer), zählt Punkte, prüft Gesamtstand.
    * **Nachrichten:**
      + S -> C: {"type": "event", "payload": {"event\_name": "round\_over", "scores\_round": {...}, "scores\_total": {...}, "double\_victory": bool}} (Broadcast)
      + Danach, wenn < 1000 Punkte: Wieder Start bei Punkt 5 (round\_start, private\_state\_update für neue Karten etc.)
16. **Punkt 18 (Spielende):**
    * **Kommunikation:** Server erkennt Spielende (>= 1000 Punkte).
    * **Nachrichten:**
      + S -> C: {"type": "event", "payload": {"event\_name": "game\_over", "winner\_team\_index": 0/1, "final\_scores": {...}}} (Broadcast)
      + **Optional:** Was passiert danach?
        - S -> C: {"type": "event", "payload": {"event\_name": "return\_to\_lobby"}} (Broadcast, wenn ein Lobby-System existiert)
        - Oder der Server startet automatisch ein neues Spiel (zurück zu Punkt 5 nach kurzer Pause?) - dann round\_start etc.

**Zusätzliche Nachrichten / Aktionen:**

* **Wunsch (Mah Jong):**
  + S -> C: {"type": "request\_action", "payload": {"request": "make\_wish"}} (Unicast an Spieler nach Spielen von Mah Jong)
  + C -> S: {"action": "make\_wish", "payload": {"value": 8}} (Wert 2-14)
  + S -> C: {"type": "event", "payload": {"event\_name": "player\_wished", "player\_index": X, "value": 8}} (Broadcast)
* **Drachen abgeben:**
  + S -> C: {"type": "request\_action", "payload": {"request": "gift\_dragon"}} (Unicast an Spieler, der Stich mit Drachen gewonnen hat)
  + C -> S: {"action": "gift\_dragon", "payload": {"target\_player\_index": Y}} (Y ist Index des Gegners)
  + S -> C: (Info ist implizit im stich\_won-Event enthalten, das anzeigt, wer die Punkte erhält)
* **Fehler:**
  + S -> C: {"type": "error", "payload": {"message": "Deine Aktion ist ungültig.", "details": "..."}} (Unicast)

**Zusammenfassung der Nachrichtenstruktur:**

* **Client an Server (C -> S):** Immer ein JSON-Objekt {"action": "...", "payload": {...}}.
* **Server an Client (S -> C):** Immer ein JSON-Objekt {"type": "...", "payload": {...}}.
  + type: Kann sein "public\_state\_update", "private\_state\_update", "event", "request\_action", "error", "joined\_table", "partner\_hand\_update".
  + payload: Enthält die spezifischen Daten für den jeweiligen Typ.

**Kartendarstellung:**

Wir nehmen die Kartendarstellung, die auch brettspielwelt.de verwendet:

\_cardlabels = (  
# rt gr bl sw  
"Hu", # Hund  
"Ma", # MahJong  
"R2", "G2", "B2", "S2", # 2  
"R3", "G3", "B3", "S3", # 3  
"R4", "G4", "B4", "S4", # 4  
"R5", "G5", "B5", "S5", # 5  
"R6", "G6", "B6", "S6", # 6  
"R7", "G7", "B7", "S7", # 7  
"R8", "G8", "B8", "S8", # 8  
"R9", "G9", "B9", "S9", # 9  
"RZ", "GZ", "BZ", "SZ", # 10  
"RB", "GB", "BB", "SB", # Bube  
"RD", "GD", "BD", "SD", # Dame  
"RK", "GK", "BK", "SK", # König  
"RA", "GA", "BA", "SA", # As  
"Dr", # Drache  
"Ph", # Phönix  
)

brettspielwelt.de ist etabliert und eindeutig. Wir verwenden diese 2-Zeichen-Strings für alle Karten in der Kommunikation (im payload von play\_cards, submit\_schupf\_cards, private\_state\_update, all\_hands\_update etc.).

## Kommunikationsweg

Ich möchte nochmal die Kommunikationwege zw. engine und Player besprechen.  
Der Weg des Agents ist ok: engine->agent->engine  
Der Weg des Clients ist nicht gut, da die Funktionen für die spezifischen Aktionen (Variante b) umgangen werden. Der Weg ist momentan: engine -> client -> browser->wsh->engine (wsh = websocket\_handler).  
Besser wäre: engine->client->browser>wsh->client->engine  
wsh müsste also die Nachrichten des Browsers an client weiterreichen, nicht an die Engine. Wie in meinem PoC wird das Warten der Engine auf eine Antowrt innerhalb der expliziten Funktion beendet. Bestenfalls muss die Engine keien UNterscheidung zw. AGent und Client machen.

Ah, ich verstehe jetzt dein Ziel und den Einwand! Du möchtest, dass die **GameEngine2 immer die gleichen Methoden aufruft, egal ob der Spieler ein Agent oder ein Client ist**. Der Unterschied soll darin liegen, *wie* diese Methoden im Client bzw. Agent implementiert sind.

**Dein gewünschter Kommunikationsweg für Clients:**

1. **GameEngine2:** Ruft eine spezifische Entscheidungs-Methode auf, z.B. player.combination(pub, priv, action\_space).
2. **Client-Objekt:** Diese Methode im Client wird aufgerufen. Sie tut *nicht* die Entscheidung selbst, sondern:
   * Sendet eine request\_action-Nachricht über den WebSocket an den Browser (await self.notify(...)).
   * **Wartet** auf eine Antwortnachricht vom Browser (z.B. play\_cards), die über den websocket\_handler zurückkommt.
3. **Browser:** Empfängt request\_action, Spieler interagiert, Browser sendet Aktionsnachricht (z.B. play\_cards).
4. **websocket\_handler:** Empfängt die Aktionsnachricht vom Browser.
5. **websocket\_handler:** Leitet diese Nachricht **an das Client-Objekt weiter** (nicht an die Engine!). Hierfür braucht der Handler eine Referenz auf das Client-Objekt und eine Methode im Client, um die Antwort zu empfangen (z.B. client.receive\_response(action\_data)).
6. **Client-Objekt:** Die receive\_response-Methode empfängt die Daten und signalisiert der wartenden Methode (aus Schritt 2), dass die Antwort da ist (z.B. über ein asyncio.Future oder Event, das *innerhalb* der combination-Methode erstellt wurde).
7. **Client-Objekt:** Die wartende combination-Methode wird aufgeweckt, holt sich das Ergebnis (die Aktion vom Browser) und gibt es an die GameEngine2 zurück.
8. **GameEngine2:** Erhält das Ergebnis von player.combination(...) (genau wie beim Agenten) und fährt fort.

**Vorteile dieses Ansatzes:**

* **Einheitliche Engine-Logik:** Die GameEngine2 muss tatsächlich nicht mehr unterscheiden, ob sie mit einem Client oder Agent spricht. Sie ruft immer dieselben Methoden auf (schupf, combination etc.).
* **Kapselung der Client-Kommunikation:** Die gesamte Logik zum Senden von Anfragen und Warten auf WebSocket-Antworten ist in der Client-Klasse gekapselt.
* **Saubere Trennung:** websocket\_handler leitet nur weiter, Client verwaltet die Anfrage/Antwort-Logik, Engine steuert den Spielablauf.

**Nachteile/Herausforderungen:**

* **Komplexität im Client:** Die Client-Klasse wird deutlich komplexer, da sie asyncio-Wartemechanismen (Futures/Events) für jede Entscheidungs-Methode implementieren muss.
* **Handler -> Client Referenz:** Der websocket\_handler braucht eine zuverlässige Methode, um die eingehende Nachricht dem richtigen, wartenden Client-Objekt zuzuordnen und dessen receive\_response-Methode aufzurufen. Das Client-Objekt wird ja bereits im Handler initialisiert/geholt, das passt also.
* **Interrupts:** Wie gehen wir mit Interrupts (z.B. Tichu-Ansage von Spieler Y, während Engine auf Aktion von Spieler X wartet) um? Wenn die Engine in await player.combination(...) "hängt", wie wird dieser Aufruf unterbrochen?
  + **Möglichkeit 1:** Die receive\_response-Methode im Client könnte auch über Interrupts informiert werden (z.B. durch ein globales Event oder einen direkten Aufruf von der Engine/Factory) und dann die wartende Methode mit einem speziellen Interrupt-Signal aufwecken.
  + **Möglichkeit 2:** Die Engine verwendet asyncio.wait wie in deinem PoC, wartet aber auf asyncio.create\_task(player.combination(...)) *und* auf die Interrupt-Events.